MASTER EN SCIENCES INFORMATIQUES

Implémentation du logiciel de vérification de modèle MEC avec les arbres partagés

Paquay, Renaud

Award date:
1996

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix Namur
Institut d'Informatique
Année académique 1995-1996

Implémentation du logiciel de vérification de modèle MEC avec les Arbres Partagés

Renaud PAQUAY

Promoteur: Baudouin LE CHARLIER

Mémoire présenté en vue de l'obtention du grade de
Licencié et Maître en Informatique

Remerciements

Je tiens à remercier mon promoteur le Professeur Baudouin Le Charlier pour son aide, sa disponibilité et sa patience durant la rédaction de ce mémoire, ainsi que Denis Zampuniéris pour ses conseils et son investissement personnel.

Je remercie également André Arnold, Didier Bégay et Paul Crubillé pour leur accueil et leur aide durant mon stage à Bordeaux.

Enfin, je remercie tous ceux qui, de près ou de loin, ont participé à l'élaboration de ce travail.

Préface

Dans ce mémoire, nous montrons comment une nouvelle structure de données, appelée *Arbre Partagé* (développée à l'Institut d'Informatique des Facultés Universitaires Notre-Dame de la Paix de Namur [11]), a pu être intégrée au logiciel *MEC* (développé à l'Université de Bordeaux I, en France [1]). MEC est un logiciel de vérification de modèle basé sur le formalisme des automates à états finis. Le logiciel, qui a déjà pu être appliqué à un certain nombre d'applications industrielles, souffre du problème de l'explosion combinatoire des automates synchronisés. Les Arbres Partagés, grâce à leur capacité à représenter de manière compacte des ensembles de n-uplets et à des algorithmes dédiés, vont permettre de pallier ce problème.

Voici un bref aperçu du contenu du mémoire:

Le **Chapitre 1** introduit la modélisation par automates à états finis avec le logiciel MEC. Un exemple complet de modélisation y est traité.

Le **Chapitre 2** décrit formellement le modèle sur lequel repose MEC. Les opérateurs sont également définis.

Le **Chapitre 3** présente les structures de données et les algorithmes de la version MEC existante.

Le **Chapitre 4** introduit les Arbres Partagés, décrit comment ils sont utilisés pour la représentation des automates synchronisés et explique les algorithmes dédiés pour la réalisation des opérateurs MEC.

Le **Chapitre 5** donne des détails techniques sur la dernière version du logiciel MEC, entièrement développée en C++. On y décrit surtout l'intégration des Arbres Partagés dans cette architecture.

Le **Chapitre 6** compare les temps de calcul et encombrement mémoire de la version MEC de base avec ceux de la version MEC avec Arbres Partagés. Plusieurs exemples, y compris des exemples pratiques, sont traités.

Le **Chapitre 7** conclut le mémoire.

Table des matières

Liste des figures

Liste des tables

Liste des algorithmes

Chapitre 1

Introduction

La vérification de systèmes de processus communiquants est devenue très importante dans les applications de parallélisme, telles que les programmes parallèles, les protocoles de réseaux, etc. Ce type de systèmes est assez complexe à vérifier par des méthodes non formelles. Il existe des exemples de protocoles très simples, apparemment corrects, mais qui souffrent de défauts majeurs. Ces défauts apparaissent en général dans des situations inhabituelles, imprévues, et provoquent une erreur de fonctionnement du système. On trouve des tels exemples dans [7].

La tendance actuelle pour vérifier qu'un système, ou une partie de celui-ci, est correct est de modéliser ce système dans un formalisme adéquat. Une fois le système modélisé, on utilise un logiciel ou des algorithmes qui vont vérifier certaines propriétés du système formalisé. On peut, par exemple, rechercher s'il existe des états dans lequel le système est bloqué.

Ce travail supplémentaire de formalisation est en général compensé par le temps gagné en vérification des propriétés. En effet, si on choisit de formaliser, c'est qu'il n'est pas possible d'effectuer "manuellement" les vérifications voulues, car le nombre de cas à traiter serait trop grand. On rencontre le même problème en algorithmique classique, où il est impossible de vérifier toutes les exécutions possibles d'un programme. La vérification est encore plus fastidieuse en parallélisme, à cause des problèmes de concurrence.

Une méthode formelle couramment utilisée est la modélisation par des automates à états finis. On modélise d'abord chaque composante ou entité du système (variables, algorithmes, etc) par des automates individuels. Chaque entité est décomposée en un certain nombre d'états possibles. Le passage d'un état à un autre se fait au moyen des transitions. Une fois ce travail effectué, on décrit les interactions possibles entre ces automates (par exemple, une instruction d'un algorithme modifie la valeur d'une variable).

Ensuite, un automate multiple, composé de tous les automates individuels, et

prenant en compte toutes les interactions possibles entre ces automates, peut être calculé. Cet automate multiple est composé d'un ensemble de *multi-états* et de *multi-transitions*[1] représentant tous les états et toutes les interactions possibles des automates composants. Il devient alors possible d'effectuer des vérifications de propriétés sur cet automate multiple. Par exemple, si un état n'a pas de successeur, on a démontré que le système peut arriver dans un état bloqué. En regardant la configuration du multi-état, on retrouve les états de chaque automate composant et ainsi l'état de chaque entité du système. La cause de l'erreur peut être éventuellement découverte au moyen d'un calcul de recherche des chemins qui mènent à cet état bloqué.

Avant d'aborder l'étude du logiciel MEC et d'entrer dans les détails de sa nouvelle architecture avec Arbres Partagés, il nous semble intéressant de montrer un exemple de modélisation d'un système par automates à états finis. Cet exemple servira également d'introduction aux concepts et à la syntaxe des commandes du logiciel MEC.

1.1 Modélisation d'un algorithme d'exclusion mutuelle

Pour illustrer l'utilité et la validité des résultats que l'on peut obtenir en modélisant des algorithmes, ou protocoles, à l'aide d'automates à états finis, nous considérons l'algorithme d'exclusion mutuelle de deux processus, proposé par Peterson dans [9].

Le modèle du système coopératif dans lequel les processus sont exécutés répond aux hypothèses suivantes :

- les instructions atomiques sont l'affectation d'une variable (écriture) et le test de la valeur d'une variable (lecture) ;
- les processus s'exécutent de manière asynchrone (par exemple, sur un ordinateur multi-processeurs). L'entrelacement des instructions atomiques est donc quelconque.

Le but de l'algorithme d'exclusion mutuelle est d'assurer que deux processus ne sont jamais en même temps dans leur *section critique* et qu'ils ne sont jamais bloqués pour un temps infini dans les protocoles d'entrée et de sortie de leur section critique.

[1] Un multi-état est un n-uplet composé d'états des automates composants. Une multi-transition est un n-uplet composé de transitions des automates composants.

Dans un premier temps, l'algorithme correct proposé par Peterson est modélisé et ses propriétés, vérifiées. Ensuite, un algorithme plus simple, mais incorrect (également présenté dans [9]), est modélisé. Nous montrons alors comment il est possible de détecter l'erreur dans l'algorithme.

1.1.1 Un algorithme correct

L'algorithme a besoin de trois variables partagées (`TURN`, `Q1` et `Q2`). Les processus (`P1` et `P2`) utilisent ces trois variables pour entrer et sortir de leur section critique. Voici l'algorithme pour P1 et P2 :

Algorithme pour P1

```
begin-loop
  0:  (...)            /* Section non-critique */
      Q1 := TRUE;      /* Protocole d'entrée pour P1 */
  1:  TURN := 1;
  2:  wait until Q2 = FALSE or TURN = 2;
  3:     (...)         /* Section critique */
      Q1 := FALSE;     /* Protocole de sortie pour P1 */
end-loop
```

Algorithme pour P2

```
begin-loop
  0:  (...)            /* Section non-critique */
      Q2 := TRUE;      /* Protocole d'entrée pour P2 */
  1:  TURN := 2;
  2:  wait until Q1 = FALSE or TURN = 1;
  3:     (...)         /* Section critique */
      Q2 := FALSE;     /* Protocole de sortie pour P2 */
end-loop
```

L'instruction `wait booléen` peut être simplement implémentée par une instruction `repeat /* instruction vide */ until booléen`. Cependant, dans un souci de simplification de la modélisation, nous considérons que `wait` est une instruction particulière mettant un processus au repos jusqu'à ce que `booléen` devienne vrai. Dans les algorithmes de P1 et P2, `booléen` est une expression composée de deux tests (`Q2 = FALSE` et `TURN = 2`, dans le cas de `P1`) ; l'évaluation de ces deux tests n'est *pas* atomique et l'ordre d'évaluation est indifférent.

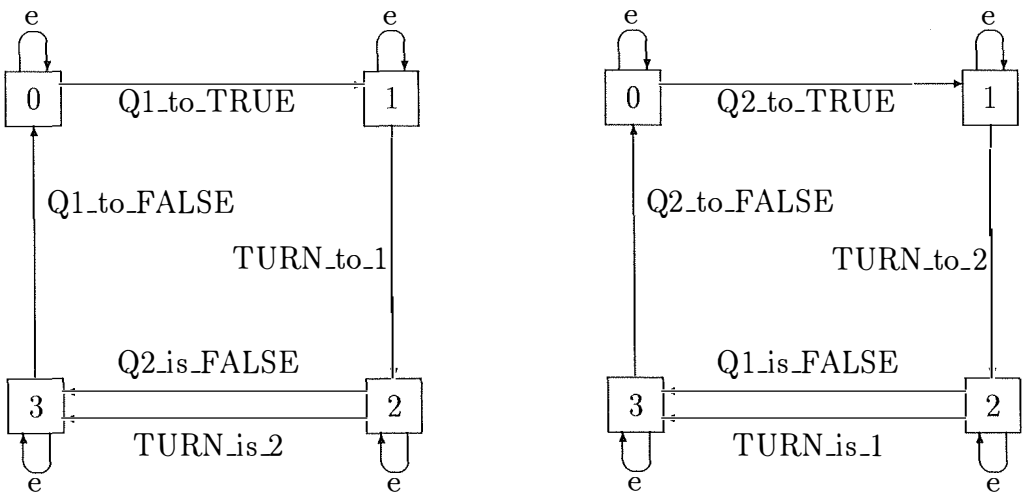


Figure 1.1: Représentation graphique des automates P1 et P2

Modélisation des automates

Les algorithmes des deux processus et les trois variables partagées sont chacun modélisés par un automate à états finis.

Automates des processus P1 et P2 Les processus P1 et P2 sont chacun modélisés par un automate à quatre états. La représentation graphique des automates P1 et P2 se trouve en Figure 1.1. Chaque état d'un automate correspond à un point d'exécution du programme (les labels '0:', '1:', '2:' et '3:' des algorithmes). Les transitions permettent de passer d'un état à l'autre de l'automate, c'est-à-dire d'un point d'exécution à un autre. Les transitions correspondent donc aux instructions d'affectation et aux tests, c'est-à-dire qu'une transition modélise une opération atomique du modèle coopératif.

La Figure 1.2 contient la définition, en langage MEC[2], des automates P1 et P2. La commande `transition_system P1` indique le commencement de la définition de l'automate `P1` (ou système de transitions `P1`, voir Chapitre 2). Pour chaque état du système, on définit les transitions applicables et l'état cible de chaque transition applicable par la notation `etat_source |- transition -> etat_cible`.

La commande `initial = {0}` définit l'ensemble des états initiaux de l'automate. Dans le cas des processus, on choisit un état initial hors de section critique (l'état 0 correspond au point d'exécution '0:' des algorithmes).

La sémantique des labels de transitions est assez explicite. Par exemple, la transition `Q1_to_TRUE` correspond à l'instruction `Q1 := TRUE`. Elle fait passer l'état

[2]La syntaxe complète du langage MEC est décrite dans [1]. La syntaxe des expressions sera étudiée plus en détail dans le Chapitre 2.

```
transition_system P1;
0 |- e           -> 0,
     Q1_to_TRUE  -> 1;
1 |- e           -> 1,
     TURN_to_1   -> 2;
2 |- e           -> 2,
     Q2_is_FALSE -> 3,
     TURN_is_2   -> 3;
3 |- e           -> 3,
     Q1_to_FALSE -> 0;
<initial = {0}>.
```

```
transition_system P2;
0 |- e           -> 0,
     Q2_to_TRUE  -> 1;
1 |- e           -> 1,
     TURN_to_2   -> 2;
2 |- e           -> 2,
     Q1_is_FALSE -> 3,
     TURN_is_1   -> 3;
3 |- e           -> 3,
     Q2_to_FALSE -> 0;
<initial = {0}>.
```

Figure 1.2: Définition MEC des automates P1 et P2

courant de l'automate de l'état '0' à l'état '1'. Les transitions de label 'e' sont définies de telle manière que l'automate reste dans son état courant[3]. Ces transitions indiquent que les processus peuvent être au repos à tout moment.

On remarque que le modèle de processus coopératifs est parfaitement respecté. En particulier, les deux conditions du `wait` ne constituent pas une opération atomique, puisqu'elles sont représentées par deux transitions distinctes. L'automate `P1` peut passer de l'état '2' à l'état '3' par la transition `Q2_is_FALSE` *ou* par la transition `TURN_is_2` (l'ordre d'exécution des transitions est quelconque).

Nous avons donc modélisé deux processus qui, dès qu'ils sortent de leur section critique, tentent immédiatement d'y entrer à nouveau. Nous faisons également l'hypothèse que les processus ne restent pas un temps infini dans leur section critique.

Automate de la variable `TURN` Pour modéliser une variable par un automate, on crée un état par valeur possible de la variable. Les transitions sont définies de manière à pouvoir changer et tester la valeur courante de la variable.

`TURN` est une variable entière dont les valeurs possibles sont '1' et '2'. Les opérations possibles sur cette variable sont l'affectation, par exemple `TURN_to_1`, et le test de la valeur courante, par exemple `TURN_is_1`. Comme dans les automates précédents, on définit des transitions de label 'e' pour chaque état. On a arbitrairement choisi 1 comme valeur initiale. La définition de `TURN` se trouve en Figure 1.3.

Automates des variables `Q1` et `Q2` Les variables `Q1` et `Q2` sont des variables booléennes. Nous modélisons les deux variables par un même automate `Q`. La

[3]Cette convention est très souvent utilisée dans le logiciel MEC pour modéliser l'asynchronisme. Cela nous sera utile lors de la définition des contraintes de synchronisation.

```
transition_system TURN;
1 |- e          -> 1,
     TURN_is_1 -> 1,
     TURN_to_1 -> 1,
     TURN_to_2 -> 2;

2 |- e          -> 2,
     TURN_is_2 -> 2,
     TURN_to_1 -> 1,
     TURN_to_2 -> 2;
<initial = {1}>.
```

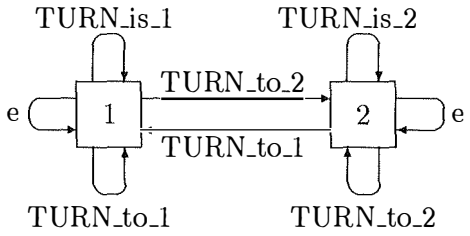


Figure 1.3: Définition MEC et représentation graphique de l'automate TURN

```
transition_system Q;
FALSE |- e           -> FALSE,
         Q_is_FALSE -> FALSE,
         Q_to_FALSE -> FALSE,
         Q_to_TRUE  -> TRUE;
TRUE  |- e           -> TRUE,
         Q_is_TRUE  -> TRUE,
         Q_to_FALSE -> FALSE,
         Q_to_TRUE  -> TRUE;
<initial = {FALSE}>.
```

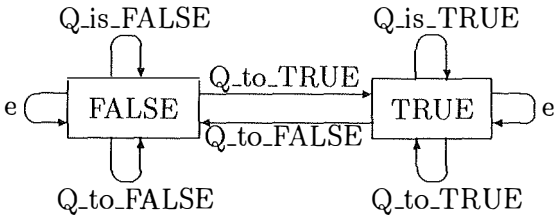


Figure 1.4: Définition MEC et représentation graphique de l'automate Q

modélisation de `Q` est semblable à celle de `TURN`, seuls les noms des états et les labels des transitions sont différents (mais la sémantique est identique). L'état initial de `Q` est `FALSE`. La définition de `Q` se trouve en Figure 1.4.

Modélisation de la synchronisation

MEC utilise une *interprétation synchrone* des automates : on suppose que chaque transition est exécutée à un instant délivré par une horloge (*clock tick*). Si deux transitions sont consécutives, alors elles sont exécutées à deux instants consécutifs de l'horloge. Il est cependant possible de modéliser un comportement asynchrone des automates en ajoutant à chaque état une transition 'boucle'. Dans les automates précédemment définis, nous avons donné le label '`e`' à ces transitions[4].

Nous avons défini les automates individuellement, il faut maintenant définir les interactions possibles entre les automates. En MEC, on fait cela au moyen d'une *contrainte de synchronisation.* La contrainte de synchronisation définit la liste des

[4]Le label '`e`' est une convention MEC.

```
synchronization_system synchro_peterson <width = 5; list =
(P1,          P2,          Q,          Q,          TURN     )>;

(Q1_to_TRUE  .e           .Q_to_TRUE  .e          .e        );
(TURN_to_1   .e           .e          .e          .TURN_to_1);
(Q2_is_FALSE .e           .e          .Q_is_FALSE .e        );
(TURN_is_2   .e           .e          .e          .TURN_is_2);
(Q1_to_FALSE .e           .Q_to_FALSE .e          .e        );

(e           .Q2_to_TRUE  .e          .Q_to_TRUE  .e        );
(e           .TURN_to_2   .e          .e          .TURN_to_2);
(e           .Q1_is_FALSE .Q_is_FALSE .e          .e        );
(e           .TURN_is_1   .e          .e          .TURN_is_1);
(e           .Q2_to_FALSE .e          .Q_to_FALSE .e        ).
```

Figure 1.5: Contrainte de synchronisation pour l'algorithme de Peterson

automates à synchroniser et se compose d'une liste de *vecteurs de synchronisation.*

Chaque vecteur de synchronisation est une séquence de labels de transitions qui définit un ensemble de transitions applicables de manière synchrone dans l'automate synchronisé. La longueur de la séquence doit être égale au nombre d'automates à synchroniser. Chaque label de transition de la séquence appartient à l'automate correspondant à la position dans la séquence. Le fait d'utiliser les labels de transitions permet de référencer plusieurs transitions dans un automate (puisqu'il peut y avoir plusieurs transitions de même label).

L'ensemble des vecteurs de synchronisation définit donc l'ensemble des transitions globales applicables de l'automate synchronisé (une définition formelle des contraintes de synchronisation se trouve dans le Chapitre 2).

Dans notre exemple, la longueur des vecteurs de synchronisation est 5, puisque nous avons 5 automates à synchroniser: `P1`, `P2`, deux fois `Q` et `TURN`. L'automate `Q` est repris deux fois dans la contrainte: le 1^{er} modélise `Q1` et le $2^{ème}$, `Q2`. La définition MEC de la contrainte de synchronisation se trouve en Figure 1.5.

La commande `synchronization_system synchro_peterson` indique le début de la définition de la contrainte de synchronisation de nom '`synchro_peterson`'. Le paramètre `width = 5` indique la taille des vecteurs de synchronisation, c'est-à-dire le nombre d'automates à synchroniser. Le paramètre `list = P1, P2, ...` indique le nom des automates à synchroniser. Ensuite, chaque vecteur de synchronisation est défini.

Par exemple, la sémantique du vecteur de synchronisation

```
(Q1_to_TRUE .e .Q_to_TRUE .e .e);
```

est la suivante :

Lorsque le 1^{er} automate (`P1`) exécute la transition `Q1_to_TRUE`, le $3^{ème}$ automate (`Q`, c'est-à-dire l'automate modélisant la variable `Q1`) exécute `Q_to_TRUE`. Les autres automates restent dans leur état courant (transition `e`). On a donc bien modélisé l'instruction *atomique* d'affectation '`Q1 := TRUE`', car les 5 transitions sont exécutées de manière synchrone[5].

Calcul du produit synchronisé

Le travail de modélisation est terminé. Le travail du logiciel MEC peut maintenant commencer. Tout d'abord, on demande le calcul de l'automate synchronisé par la commande :

```
sync(synchro_peterson,peterson);
```

où le 1^{er} paramètre est le nom de la contrainte de synchronisation et le $2^{ème}$, le nom de l'automate synchronisé.

Après calcul, l'automate synchronisé `peterson` contient l'ensemble des transitions qui respectent la contrainte de synchronisation, restreintes aux transitions et états accessibles à partir des états initiaux (voir Chapitre 2 pour une définition formelle du produit synchronisé accessible). L'ensemble des états initiaux de l'automate synchronisé est le produit cartésien des ensembles des états initiaux des automates composants. Dans notre exemple, l'automate synchronisé a un seul état initial, car chaque automate composante a un état initial.

La Figure 1.6 représente l'automate synchronisé. Il contient 20 états et 34 transitions résultant du calcul du produit synchronisé accessible.

Un état synchronisé (ou multi-état) est noté "`e(e1.e2.e3.e4.e5)`", où chaque `ei` appartient au $i^{ème}$ automate composant. Une transition synchronisée (ou multi-transition) est notée "`(t1.t2.t3.t4.t5)`", où chaque `ti` appartient au $i^{ème}$ automate composant.

L'état initial est `e(0.0.FALSE.FALSE.1)`. Deux transitions peuvent être appliquées à cet état : `(e.Q2_to_TRUE.e.Q_to_TRUE.e)` et `(Q1_to_TRUE.e.Q_to_TRUE.e.e)`. Elles correspondent aux deux entrelacements possibles de la première instruction de `P1` et de `P2`. Ces deux transitions mènent à deux nouveaux états, auxquels de nouvelles transitions peuvent être appliquées, et ainsi de suite jusqu'à ce qu'aucune transition ne puisse être

[5] Ce vecteur représente exactement 32 transitions globales applicables de l'automate synchronisé : la transition `Q1_to_TRUE` est présente une fois dans `P1`, la transition `e` est présente quatre fois dans `P2`, `Q_to_TRUE` est présente deux fois dans `Q`, etc. Le nombre de transitions représenté est donc $1 * 4 * 2 * 2 * 2 = 32$.

```
transition_system peterson < width = 5;  list = (P1, P2, Q, Q, TURN )>;
e(0.0.FALSE.FALSE.1) |-(e.Q2_to_TRUE.e.Q_to_TRUE.e)   -> e(0.1.FALSE.TRUE.1),
                       (Q1_to_TRUE.e.Q_to_TRUE.e.e)   -> e(1.0.TRUE.FALSE.1);
e(0.1.FALSE.TRUE.1)  |-(e.TURN_to_2.e.e.TURN_to_2)    -> e(0.2.FALSE.TRUE.2),
                       (Q1_to_TRUE.e.Q_to_TRUE.e.e)   -> e(1.1.TRUE.TRUE.1);
e(1.0.TRUE.FALSE.1)  |-(e.Q2_to_TRUE.e.Q_to_TRUE.e)   -> e(1.1.TRUE.TRUE.1),
                       (TURN_to_1.e.e.e.TURN_to_1)    -> e(2.0.TRUE.FALSE.1);
e(0.2.FALSE.TRUE.2)  |-(e.Q1_is_FALSE.Q_is_FALSE.e.e) -> e(0.3.FALSE.TRUE.2),
                       (Q1_to_TRUE.e.Q_to_TRUE.e.e)   -> e(1.2.TRUE.TRUE.2);
e(1.1.TRUE.TRUE.1)   |-(e.TURN_to_2.e.e.TURN_to_2)    -> e(1.2.TRUE.TRUE.2),
                       (TURN_to_1.e.e.e.TURN_to_1)    -> e(2.1.TRUE.TRUE.1);
e(2.0.TRUE.FALSE.1)  |-(e.Q2_to_TRUE.e.Q_to_TRUE.e)   -> e(2.1.TRUE.TRUE.1),
                       (Q2_is_FALSE.e.e.Q_is_FALSE.e) -> e(3.0.TRUE.FALSE.1);
e(0.3.FALSE.TRUE.2)  |-(e.Q2_to_FALSE.e.Q_to_FALSE.e) -> e(0.0.FALSE.FALSE.2),
                       (Q1_to_TRUE.e.Q_to_TRUE.e.e)   -> e(1.3.TRUE.TRUE.2);
e(1.2.TRUE.TRUE.2)   |-(TURN_to_1.e.e.e.TURN_to_1)    -> e(2.2.TRUE.TRUE.1);
e(2.1.TRUE.TRUE.1)   |-(e.TURN_to_2.e.e.TURN_to_2)    -> e(2.2.TRUE.TRUE.2);
e(3.0.TRUE.FALSE.1)  |-(e.Q2_to_TRUE.e.Q_to_TRUE.e)   -> e(3.1.TRUE.TRUE.1),
                       (Q1_to_FALSE.e.Q_to_FALSE.e.e) -> e(0.0.FALSE.FALSE.1);
e(0.0.FALSE.FALSE.2) |-(e.Q2_to_TRUE.e.Q_to_TRUE.e)   -> e(0.1.FALSE.TRUE.2),
                       (Q1_to_TRUE.e.Q_to_TRUE.e.e)   -> e(1.0.TRUE.FALSE.2);
e(1.3.TRUE.TRUE.2)   |-(e.Q2_to_FALSE.e.Q_to_FALSE.e) -> e(1.0.TRUE.FALSE.2),
                       (TURN_to_1.e.e.e.TURN_to_1)    -> e(2.3.TRUE.TRUE.1);
e(2.2.TRUE.TRUE.1)   |-(e.TURN_is_1.e.e.TURN_is_1)    -> e(2.3.TRUE.TRUE.1);
e(2.2.TRUE.TRUE.2)   |-(TURN_is_2.e.e.e.TURN_is_2)    -> e(3.2.TRUE.TRUE.2);
e(3.1.TRUE.TRUE.1)   |-(e.TURN_to_2.e.e.TURN_to_2)    -> e(3.2.TRUE.TRUE.2),
                       (Q1_to_FALSE.e.Q_to_FALSE.e.e) -> e(0.1.FALSE.TRUE.1);
e(0.1.FALSE.TRUE.2)  |-(e.TURN_to_2.e.e.TURN_to_2)    -> e(0.2.FALSE.TRUE.2),
                       (Q1_to_TRUE.e.Q_to_TRUE.e.e)   -> e(1.1.TRUE.TRUE.2);
e(1.0.TRUE.FALSE.2)  |-(e.Q2_to_TRUE.e.Q_to_TRUE.e)   -> e(1.1.TRUE.TRUE.2),
                       (TURN_to_1.e.e.e.TURN_to_1)    -> e(2.0.TRUE.FALSE.1);
e(2.3.TRUE.TRUE.1)   |-(e.Q2_to_FALSE.e.Q_to_FALSE.e) -> e(2.0.TRUE.FALSE.1);
e(3.2.TRUE.TRUE.2)   |-(Q1_to_FALSE.e.Q_to_FALSE.e.e) -> e(0.2.FALSE.TRUE.2);
e(1.1.TRUE.TRUE.2)   |-(e.TURN_to_2.e.e.TURN_to_2)    -> e(1.2.TRUE.TRUE.2),
                       (TURN_to_1.e.e.e.TURN_to_1)    -> e(2.1.TRUE.TRUE.1);
< initial = { e(0.0.FALSE.FALSE.1) } >.
```

Figure 1.6: Automate synchronisé de Peterson

appliquée ou que les transitions appliquées mènent à des états déjà présents dans l'automate synchronisé.

Vérification des propriétés

On peut maintenant effectuer le calcul des propriétés de l'automate synchronisé. Les propriétés que nous allons vérifier sont le respect de l'exclusion mutuelle, l'absence d'interblocage et l'absence de *livelock*.

Exclusion mutuelle L'algorithme ne doit pas permettre que les deux processus soient en même temps dans leur section critique. Il suffit de vérifier qu'il n'existe aucun multi-état de l'automate synchronisé tel que les 1^{er} et $2^{ème}$ états composants soient tous deux égaux à '3'. On peut vérifier cela par la commande MEC :

```
P1_et_P2_dans_CS := (!state[1] = ´3´ /\ !state[2] = ´3´);
```

où `!state[`i`] = ´3´` calcule l'ensemble des multi-états de l'automate synchronisé dont la $i^{ème}$ composante a le nom `´3´` ; `/\` est l'opérateur ensembliste d'intersection de deux ensembles.

Les résultat de l'évaluation est stocké dans l'ensemble `P1_et_P2_dans_CS`[6]. Cet ensemble ne contient aucun état, on peut donc conclure que l'exclusion mutuelle est respectée dans *tous les cas possibles d'exécution des 2 processus.*

Absence d'interblocage Etant donné que nous avons modélisé deux processus qui tentent continuellement d'entrer en section critique et que nous avons fait l'hypothèse qu'ils ne restent pas un temps infini dans leur section critique, on peut vérifier qu'il n'existe pas de situations d'interblocage. Nous appelons situation d'interblocage une situation dans laquelle les deux processus sont mis en attente infinie dans l'instruction `wait`. Cette instruction constitue en effet le point "critique" de l'algorithme d'exclusion mutuelle.

Une situation d'interblocage se traduit, dans l'automate synchronisé, par la présence d'un ou plusieurs multi-états qui n'ont pas de successeurs, c'est-à-dire qui ne sont la source d'aucune transition. Nous appelons *deadlock* l'ensemble de ces multi-états. La commande MEC qui permet de calculer l'ensemble *deadlock* est :

```
deadlock := * - src(*);
```

[6]Dans la terminologie MEC, un sous-ensemble des états ou des transitions d'un système de transitions est appelé *marque*.

où * est l'ensemble de tous les états ou de toutes les transitions du système (suivant le contexte de l'évaluation) ; - est l'opérateur ensembliste de différence ; src(T) calcule l'ensemble des états du système qui sont la source d'au moins une transition de T. L'ensemble deadlock est vide, ce qui indique l'absence d'interblocage.

Remarque

Si nous avions modélisé l'instruction `wait booléen` par un boucle d'attente `repeat /* instruction vide */ until booléen`, la recherche d'interblocage par le calcul de l'ensemble `deadlock` n'est pas suffisante, car les processus ne sont jamais inactifs. L'ensemble aurait donc de toute façon été vide. Nous allons vérifier, dans la section suivante, l'absence d'une situation d'interblocage plus complexe : le *livelock*.

Cependant, de manière générale, il est intéressant de calculer `deadlock`, car la présence d'éléments imprévus dans cet ensemble peut non seulement indiquer une erreur de conception, mais aussi une erreur de modélisation.

Absence de *livelock* Nous appelons *livelock* une exécution infinie dans laquelle les deux processus P1 et P2 essayent d'entrer dans leur section critique, mais n'y parviennent jamais[8]. Une situation de *livelock* est plus complexe qu'une situation d'interblocage, car la présence d'un *livelock* signifie que les deux processus sont incapables d'entrer en section critique, tous en restant actifs, c'est-à-dire sans être bloqués dans l'instruction `wait`. Les processus bouclent donc constamment autour de leur section critique, sans jamais pouvoir y entrer.

Les commandes MEC suivantes calculent l'ensemble des transitions du système dans lesquelles les deux processus sont actifs :

```
P1_actif := !label[1] # 'e';
P2_actif := !label[2] # 'e';
```

où `!label[`i`] # 'e'` est l'ensemble des multi-transitions du système dont la $i^{ème}$ composante n'est pas libellée par 'e'.

Les commandes MEC suivantes calculent l'ensemble des transitions dans lesquelles les deux processus essayent d'entrer dans leur section critique :

```
P1_essaye := rsrc(!state[1] # '3');
P2_essaye := rsrc(!state[2] # '3');
P1_et_P1_essayent != P1_essaye /\ P2_essaye;
```

où rsrc(S) est l'ensemble des transitions qui ont leur origine dans l'ensemble des états de S. La commande MEC suivante calcule l'ensemble des transitions qui forment un *livelock* :

```
livelock := loop(P1_actif, P1_et_P2_essayent) /\
            loop(P2_actif, P1_et_P2_essayent);
```

où `loop`(R, R') est l'ensemble des transitions appartenant à un chemin non-vide $t_1, \ldots, t_n$ tel que l'origine de t_1 est égale à la cible de t_n, $t_j \in R'$ pour tout $j = 1..n$ et $t_i \in R$ pour au moins un $i \in \{1..n\}$. Nous reviendrons sur l'opérateur `loop` dans le Chapitre 2.

Le premier opérateur calcule l'ensemble des transitions telles que P1 est actif, mais n'arrive jamais à entrer en section critique. Le $2^{ème}$ opérateur fait de même pour le processus P2. L'intersection des deux ensembles obtenus donne l'ensemble `livelock`, qui est vide.

1.1.2 Un algorithme incorrect

L'algorithme proposé dans cette section est une version simplifiée de l'algorithme d'exclusion mutuelle vu dans la section précédente. L'algorithme est identique excepté que la variable `TURN` n'est plus utilisée. Nous allons démontrer que cette version de l'algorithme est incorrecte.

Algorithme pour P1

```
begin-loop
  0:  (...)            /* Section non-critique */
      Q1 := TRUE;      /* Protocole d'entrée pour P1 */
  1:  wait until Q2 = FALSE;
  2:     (...)         /* Section critique */
      Q1 := FALSE;     /* Protocole de sortie pour P1 */
end-loop
```

Algorithme pour P2

```
begin-loop
  0:  (...)            /* Section non-critique */
      Q2 := TRUE;      /* Protocole d'entrée pour P1 */
  1:  wait until Q1 = FALSE;
  2:     (...)         /* Section critique */
      Q2 := FALSE;     /* Protocole de sortie pour P1 */
end-loop
```

En examinant attentivement les deux algorithmes, on peut immédiatement détecter le problème: si les deux processus se trouvent même temps en '`1:`',

```
transition_system P1;
0 |- e           -> 0,
     Q1_to_TRUE  -> 1;
1 |- e           -> 1,
     Q2_is_FALSE -> 2;
2 |- e           -> 2,
     Q1_to_FALSE -> 0;
<initial = {0}>.
```

```
transition_system P2;
0 |- e           -> 0,
     Q2_to_TRUE  -> 1;
1 |- e           -> 1,
     Q1_is_FALSE -> 2;
2 |- e           -> 2,
     Q2_to_FALSE -> 0;
<initial = {0}>.
```

Figure 1.7: Définition MEC des automates P1 et P2 (version incorrecte)

```
synchronization_system pet_bug <width = 4; list =
(P1,          P2,          Q,          Q          )>;

(Q1_to_TRUE  .e           .Q_to_TRUE  .e          );
(Q2_is_FALSE .e           .e          .Q_is_FALSE );
(Q1_to_FALSE .e           .Q_to_FALSE .e          );

(e           .Q2_to_TRUE  .e          .Q_to_TRUE  );
(e           .Q1_is_FALSE .Q_is_FALSE .e          );
(e           .Q2_to_FALSE .e          .Q_to_FALSE ).
```

Figure 1.8: Contrainte de synchronisation pour l'algorithme de Peterson (version incorrecte)

les variables `Q1` et `Q2` sont toutes les deux à 'TRUE', ce qui implique que `P1` et `P2` sont en attente infinie dans le `wait`. Nous allons montrer comment il est possible de détecter cette erreur de manière formelle.

Modélisation des automates

La modélisation de `P1` et `P2` est similaire à celle vue dans la section précédente (Figure 1.7). La variable `TURN` n'est plus prise en compte. La variable `Q` est modélisée exactement de la même façon.

Modélisation de la synchronisation

La contrainte de synchronisation (Figure 1.8) ne comporte plus que quatre automates (`P1`, `P2`, et deux fois `Q`).

Calcul du produit synchronisé

L'automate synchronisé (Figure 1.9) contient 8 états et 12 transitions.

```
transition_system pet_bug < width = 4;  list = (P1, P2, Q, Q )>;

e(0.0.FALSE.FALSE) |-
        (e.Q2_to_TRUE.e.Q_to_TRUE)   -> e(0.1.FALSE.TRUE),
        (Q1_to_TRUE.e.Q_to_TRUE.e)   -> e(1.0.TRUE.FALSE);

e(0.1.FALSE.TRUE)  |-
        (e.Q1_is_FALSE.Q_is_FALSE.e) -> e(0.2.FALSE.TRUE),
        (Q1_to_TRUE.e.Q_to_TRUE.e)   -> e(1.1.TRUE.TRUE);

e(1.0.TRUE.FALSE)  |-
        (e.Q2_to_TRUE.e.Q_to_TRUE)   -> e(1.1.TRUE.TRUE),
        (Q2_is_FALSE.e.e.Q_is_FALSE) -> e(2.0.TRUE.FALSE);

e(0.2.FALSE.TRUE)  |-
        (e.Q2_to_FALSE.e.Q_to_FALSE) -> e(0.0.FALSE.FALSE),
        (Q1_to_TRUE.e.Q_to_TRUE.e)   -> e(1.2.TRUE.TRUE);

e(1.1.TRUE.TRUE)   |- ;

e(2.0.TRUE.FALSE)  |-
        (e.Q2_to_TRUE.e.Q_to_TRUE)   -> e(2.1.TRUE.TRUE),
        (Q1_to_FALSE.e.Q_to_FALSE.e) -> e(0.0.FALSE.FALSE);

e(1.2.TRUE.TRUE)   |-
        (e.Q2_to_FALSE.e.Q_to_FALSE) -> e(1.0.TRUE.FALSE);

e(2.1.TRUE.TRUE)   |-
        (Q1_to_FALSE.e.Q_to_FALSE.e) -> e(0.1.FALSE.TRUE);
<initial = { e(0.0.FALSE.FALSE) }>.
```

Figure 1.9: Automate synchronisé de Peterson (version incorrecte)

Vérification des propriétés

La vérification des propriétés du système est faite au moyen des mêmes commandes que précédemment.

Exclusion mutuelle La commande

```
P1_et_P2_dans_CS := (!state[1] = '3' /\ !state[2] = '3');
```

donne comme résultat un ensemble vide. L'algorithme respecte donc la propriété d'exclusion mutuelle.

Absence d'interblocage La commande

```
deadlock := * - src(*);
```

donne comme résultat un ensemble contenant l'état `e(1.1.TRUE.TRUE)`. Ce résultat correspond bien à la constatation intuitive que l'on avait de l'erreur: lorsque `P1` et `P2` sont en même temps en '`1:`', `Q1` et `Q2` valent tous deux `TRUE` et les deux processus sont en attente infinie. Il devient donc inutile de vérifier la propriété d'absence de *livelock*.

En conclusion de cette introduction à la modélisation, nous avons vu comment il est possible de modéliser des algorithmes et des variables au moyen d'automates à états finis. Nous avons également vu quelques-uns des opérateurs du logiciel MEC, qui permettent la vérification de toutes sortes de propriétés sur ces automates.

D'un point de vue théorique, tout système informatique (algorithmes, variables, protocoles, etc) est modélisable sous forme d'automate, puisqu'un ordinateur est lui-même théoriquement modélisable par un automate.

Un problème pratique que l'on rencontre est l'explosion combinatoire de l'ensemble des états et transitions des automates synchronisés. En effet, le taille des automates synchronisés a tendance à croître exponentiellement avec le nombre d'automates composants. Dans l'exemple de l'algorithme de Peterson que nous avons considéré, l'automate synchronisé ne comportait que quelques dizaines d'états et transitions. Si on modélise le même algorithme avec 4 processus au lieu de 2, il faut 11 automates (4 processus, 4 variables '`Q`' et 3 variables '`TURN`'). La taille de l'automate synchronisé monte alors à quelques dizaines de *milliers* d'états et transitions. Dans le chapitre Chapitre 6, nous montrerons des exemples d'automates ayant jusqu'à 10^{12} états et transitions, et même au delà.

Les limitations de la version actuelle du logiciel MEC, qui utilise une représentation explicite des automates (c'est-à-dire que chaque état et chaque

transition est représenté individuellement), sont essentiellement dues à des problèmes d'encombrement mémoire et de temps de calcul. En effet, le nombre d'états et de transitions d'un automate synchronisé augmentant exponentiellement avec le nombre d'automates composants, la représentation explicite devient très rapidement trop gourmande en mémoire. De plus, tous les algorithmes de MEC sont linéaires sur le nombre d'états et de transitions d'un automate, ce qui implique des temps de calcul également exponentiel sur le nombre d'automates composants.

C'est pourquoi il semblait judicieux d'introduire une nouvelle structure de données, les Arbres Partagés, capable de stocker des ensembles de n-uplets de manière très compacte, et d'effectuer des opérations globales sur ces ensembles. Le sujet de ce mémoire est de montrer comment cela a pu être réalisé.

1.2 Avantages et inconvénients de la modélisation par automates

Les avantages ont été illustrés dans les sections précédentes :

- Les concepts manipulés pour la modélisation sont simples (états, transitions, synchronisation).
- Tout système informatique est, en théorie, modélisable par un automate.
- Il est possible de vérifier systématiquement toute une série de propriétés du problème modélisé.

Il existe malheureusement des inconvénients inhérents à ce type de modélisation :

- La taille des automates synchronisé a tendance à croître exponentiellement avec le nombre d'automates à synchroniser.
- Enfin, si on peut effectivement vérifier systématiquement toutes sortes de propriétés, il n'est pas toujours évident que la modélisation du problème est correcte. Dans le cas où la modélisation est incorrecte, les résultats le seront également. De plus, la structure 'plate' des automates rend le travail de modélisation encore plus complexe.

Chapitre 2

MEC

Dans ce chapitre, le modèle sur lequel repose le logiciel MEC est défini formellement. Ces définitions sont reprises de [1] et [4]. Dans un premier temps, les objets manipulés sont définis, ensuite les opérateurs du logiciel permettant la manipulation de ces objets sont spécifiés.

2.1 Modèle

Le modèle MEC repose principalement sur les *systèmes de transitions*, qui correspondent aux automates, et sur les *contraintes de synchronisation*, qui définissent les interactions entre les automates.

2.1.1 Système de transitions

Un *système de transitions libellé* sur un alphabet A d'*actions* ou *événements* est un n-uplet $\mathcal{A} = \langle S, T, \alpha, \beta, \lambda \rangle$, où

- S est un ensemble fini d'états,
- T est un ensemble fini de transitions,
- $\alpha, \beta : T \to S$ sont des applications qui associent à chaque transition t son état source $\alpha(t)$ et son état cible $\beta(t)$,
- $\lambda : T \to A$ est un application qui associe à chaque transition t l'action ou l'événement $\lambda(t)$ qui est la cause de la transition.

Un *système de transitions paramètré* est un système de transitions avec des marques d'états et de transitions. On appelle *marque d'états* un sous-ensemble M de S; les états appartenant à M seront des états marqués M. De même, les

marques de transitions sont des sous-ensembles de T. La marque d'état *initial* est une marque obligatoire qui définit l'ensemble des états initiaux du système de transitions.

Un *chemin* de longueur $n > 0$ dans un système de transitions $\mathcal{A} = \langle S, T, \alpha, \beta, \lambda \rangle$ est une séquence de transitions $p = t_1, \ldots, t_n$ telle que pour tout $i = 1, \ldots, n-1$, $\beta(t_i) = \alpha(t_{i+1})$. La source du chemin p, notée $\alpha(p)$, est $\alpha(t_1)$. La cible du chemin p, notée $\beta(p)$, est $\beta(t_n)$.

2.1.2 Système de transitions synchronisé

Comme nous l'avons dit dans le Chapitre 1, MEC utilise une *interprétation synchrone* des systèmes de transitions : on suppose que chaque transition est exécutée à un instant délivré par une horloge (*clock tick*). Si deux transitions sont consécutives, alors elles sont exécutées à deux instants consécutifs de l'horloge. Il est cependant possible de modéliser un comportement asynchrone des systèmes de transitions en ajoutant à chaque état une transition 'boucle' (la transition '`e`').

Contrainte de synchronisation

Les interactions entre les systèmes de transitions $\mathcal{A}_i$, définis respectivement sur les alphabets A_i, d'un système sont représentées par un sous-ensemble I de $A_1 \times \cdots \times A_n$. Ce sous-ensemble est appelé *contrainte de synchronisation*. Un exemple de contrainte de synchronisation a été donné dans le Chapitre 1.

Produit synchronisé

Etant donné un vecteur $\langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ de systèmes de transitions tel que chaque $\mathcal{A}_i = \langle S_i, T_i, \alpha_i, \beta_i, \lambda_i \rangle$ est défini sur l'alphabet A_i, le *produit libre* de $\langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ est un système de transitions $\langle S, T, \alpha, \beta, \lambda \rangle$ défini sur l'alphabet $A_1 \times \cdots \times A_n$ tel que :

- $S = S_1 \times \cdots \times S_n$,
- $T = T_1 \times \cdots \times T_n$,
- $\alpha(t_1, \ldots, t_n) = \langle \alpha_1(t_1), \ldots, \alpha_n(t_n) \rangle$,
- $\beta(t_1, \ldots, t_n) = \langle \beta_1(t_1), \ldots, \beta_n(t_n) \rangle$,
- $\lambda(t_1, \ldots, t_n) = \langle \lambda_1(t_1), \ldots, \lambda_n(t_n) \rangle$.

Etant donné une contrainte de synchronisation I incluse dans $A_1 \times \cdots \times A_n$, le *produit synchronisé* de $\langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ sur I est le système de transitions $\langle S, T_I, \alpha, \beta, \lambda \rangle$ tel que :

- $\langle S, T, \alpha, \beta, \lambda \rangle$ est le produit libre de $\mathcal{A}_1, \ldots, \mathcal{A}_n$,
- T_I est l'ensemble des transitions $t = \langle t_1, \ldots, t_n \rangle$ de T dont les labels $\lambda(t) = \langle \lambda_1(t_1), \ldots, \lambda_n(t_n) \rangle$ appartiennent à I.

MEC prend en compte seulement un sous-système de transitions du produit synchronisé : la partie accessible de celui-ci.

L'ensemble des états du système de transitions synchronisé est

$$initial \cup Reach(initial)$$

avec

- $initial = initial_1 \times \cdots \times initial_n$, où $initial_i$ est l'ensemble des états initiaux de $\mathcal{A}_i$,
- $Reach(initial)$ est l'ensemble des états accessibles à partir de l'ensemble $initial$.

L'ensemble des transitions du système de transitions synchronisé est le sous-ensemble de l'ensemble des transitions globales ayant leur source et leur cible dans l'ensemble des états du système.

2.2 Calcul de propriétés

Le langage de commande de MEC est basé sur l'exécution d'instructions du type :

```
variable := expression;
```

où `variable` est le nom d'une marque et `expression`, une expression ensembliste valide du langage MEC. La valeur de `expression` est calculée et assignée à la marque `variable`. La valeur de l'expression est soit un ensemble d'états, soit un ensemble de transitions.

Nous n'analysons pas en détail la syntaxe des expressions MEC (la syntaxe complète est décrite dans [1]). Nous nous contentons de citer les opérateurs et de les définir formellement.

En plus des opérateurs définis dans les sections suivantes, MEC accepte les opérateurs ensemblistes union (notée `\/`), intersection (notée `/\`) et différence (notée `-`), la constante "ensemble vide" (notée `{}`) et les constantes "ensemble de tous les états" ou "ensemble de toutes les transitions" (toutes deux notées `*`). Le

nom d'une marque dans une expression est considéré comme une constante, dont la valeur est la valeur de la marque.

Avant d'évaluer une expression, MEC effectue une analyse sémantique pour vérifier que le type ("états" ou "transitions") des arguments, variables, opérateurs, ... composant l'expression est correct.

2.2.1 Opérateurs élémentaires

Etant donné un système de transitions $\mathcal{A} = \langle S, T, \alpha, \beta, \lambda \rangle$, Q un sous-ensemble de S et R un sous-ensemble de T, les quatre opérateurs élémentaires sont :

$$\begin{aligned} \texttt{src}(R) &= \{\alpha(t) \mid t \in R\} \\ \texttt{tgt}(R) &= \{\beta(t) \mid t \in R\} \\ \texttt{rsrc}(Q) &= \{t \mid \alpha(t) \in Q\} \\ \texttt{rtgt}(Q) &= \{t \mid \beta(t) \in Q\} \end{aligned}$$

2.2.2 Composantes fortement connexes

L'opérateur de calcul des composantes fortement connexes est l'opérateur `loop`.

Etant donné un système de transitions $\mathcal{A} = \langle S, T, \alpha, \beta, \lambda \rangle$ et R, R' des sous-ensembles de T, une transition $t \in \texttt{loop}(R, R')$ si et seulement si t appartient à un chemin non-vide p tel que :

1. la source de p est égale à sa cible ($\alpha(p) = \beta(p)$),
2. chaque transition de p est dans R',
3. au moins une transition de p est dans R.

On peut donc dire qu'une transition $t \in \texttt{loop}(R, R')$ si elle fait partie d'une composante fortement connexe du graphe de R', chaque composante contenant au moins une transition de R.

2.2.3 Plus courts chemins

L'opérateur de calcul des plus courts chemins est l'opérateur `trace`.

Etant donné un système de transitions $\mathcal{A} = \langle S, T, \alpha, \beta, \lambda \rangle$, Q_1, Q_2 des sous-ensembles de S et R un sous-ensemble de T, $\texttt{trace}(Q_1, R, Q_2)$ est un plus court chemin non-vide p tel que $\alpha(p) \in Q_1$ et $\beta(p) \in Q_2$ et tel que, pour toute transition $t \in p$, $t \in R$.

L'opérateur `trace` permet donc de rechercher un plus court chemin reliant deux ensembles d'états, chemin dont toutes les transitions appartiennent à un ensemble de transitions donné. Il peut évidemment exister plusieurs plus courts chemins répondant à la définition de `trace`. La définition MEC ne précise pas comment le choix d'un chemin plutôt qu'un autre est fait.

2.2.4 Accessibilité et co-accessibilité

L'opérateur de calcul d'accessibilité est l'opérateur `reach`.

Etant donné un système de transitions $\mathcal{A} = \langle S, T, \alpha, \beta, \lambda \rangle$, Q un sous-ensemble de S et R un sous-ensemble de T, un état $e \in \texttt{reach}(Q, R)$ si et seulement si il existe un chemin p tel que $\alpha(p) \in Q$ et $\beta(p) = e$ et tel que, pour toute transition $t \in p$, $t \in R$.

L'opérateur `reach` calcule donc l'ensemble des états accessibles à partir d'un ensemble d'états donné, R étant une restriction sur l'ensemble des transitions à parcourir.

L'opérateur de calcul de co-accessibilité est l'opérateur `coreach`.

Un état $e \in \texttt{coreach}(Q, R)$ si et seulement si il existe un chemin p tel que $\alpha(p) = e$ et $\beta(p) \in Q$ et tel que, pour toute transition $t \in p$, $t \in R$.

L'opérateur `coreach` calcule donc l'ensemble des états co-accessibles à partir d'un ensemble d'états donné, R étant une restriction sur l'ensemble des transitions à parcourir.

2.2.5 Comparaisons de labels

MEC permet l'évaluation de toute une série d'expressions sur les noms des états et les labels des transitions. Nous avons vu des exemples d'utilisation des ces expressions dans le Chapitre 1. Voici la liste des expressions de "comparaisons de labels" :

`!state = PATTERN` **ou** `!state # PATTERN`

La valeur de l'expression est un sous-ensemble de S tel que le nom de chaque état est égal ('=') ou différent ('#') à la chaîne `PATTERN`.

Exemple

Dans l'automate synchronisé de Peterson vu en Section 1.1.1, l'expression `!state = ´*3*´` retourne l'ensemble des états tels que le processus `P1` ou `P2` est en section critique[1].

[1] Le caractère '*' désigne une séquence quelconque de caractères.

`!label = PATTERN` **ou** `!label # PATTERN`
Idem pour les labels de transitions.

`!state[n] = PATTERN` **ou** `!state[n] # PATTERN`
Si $\mathcal{A}$ est un système de transitions synchronisé, la valeur de l'expression est un sous-ensemble de S tel que le nom du $n^{ème}$ état composant est égal ('=') ou différent ('#') à la chaîne `PATTERN`.

Exemple
Dans l'automate synchronisé de Peterson vu en Section 1.1.1, l'expression `!state[1] = ´3´` retourne l'ensemble des états tels que le processus `P1` est en section critique.

`!label[n] = PATTERN` **ou** `!label[n] # PATTERN`
Idem pour les labels de transitions.

`!state[n] = !state[m]` **ou** `!state[n] # !state[m]`
Si $\mathcal{A}$ est un système de transitions synchronisé, la valeur de l'expression est un sous-ensemble de S tel que le nom du $n^{ème}$ état composant est égal ('=') ou différent ('#') au nom du $m^{ème}$ état composant.

Exemple
Dans l'automate synchronisé de Peterson vu en Section 1.1.1, l'expression `!state[1] = !state[2]` retourne l'ensemble des états tels que les processus `P1` et `P2` se trouvent en même temps en des points d'exécution de noms identiques.

`!label[n] = !label[m]` **ou** `!label[n] # !label[m]`
Idem pour les labels de transitions.

'`!state`' est un mot-clé donnant le type 'états' à l'expression. '`!label`' est un mot-clé donnant le type 'transitions' à l'expression. '`PATTERN`' désigne une chaîne de caractères pouvant contenir des jokers ("*wildcards*"). Les jokers reconnus sont le point d'interrogation ('?'), qui désigne un caractère quelconque, et l'astérisque ('*'), qui désigne une séquence quelconque (y compris vide) de caractères.

2.2.6 Appartenance à des sous-marques

Etant donné un système de transitions synchronisé, l'expression `sous_marque[`i`]` désigne un sous-ensemble des n-uplets de S ou T (suivant le type de `sous_marque`) dont la $i^{ème}$ composante appartient à la marque `sous_marque`, définie sur le $i^{ème}$ système de transitions composant.

2.3 Définition de nouveaux opérateurs

Cette section est entièrement basée sur [4], pages 18 à 29.

Le langage de commande de MEC offre la possibilité de définir des nouvelles fonctions (ou opérateurs) comme plus petite solution d'un système d'équations.

Pour pouvoir intégrer ces opérateurs dans le calcul des expressions MEC, seule une variable du système sera le résultat de la fonction, les autres ne servant qu'à établir cette valeur. Le nombre et le type des paramètres de la fonction est défini par l'utilisateur.

MEC calcule la valeur de la variable comme étant le point fixe d'une fonction monotone. Un critère syntaxique sur les équations permet de garantir la monotonie des équations et l'existence d'un plus petit (ou plus grand) point fixe.

Le membre droit (appelé *terme*) d'une équation est une expression MEC limitée à certains opérateurs :

Les *termes* sont formés à partir des opérateurs binaires d'union, d'intersection, et de différence ensembliste, des appels de fonctions élémentaires (`src`, `tgt`, `rsrc`, `rtgt`) et d'identificateurs désignant des variables ou des paramètres d'états ou de transitions ainsi que de l'ensemble vide (`{}`) et des ensembles de tous les états et de toutes les transitions (`*`). Les variables et paramètres sont des marques d'états ou de transitions.

2.3.1 Monotonie syntaxique des termes

On peut montrer que tous les opérateurs qui composent les termes sont monotones, sauf la différence ensembliste, qui est monotone par rapport à son 1^{er} argument et anti-monotone par rapport à son $2^{ème}$ argument. Ainsi, pour tout ensemble A, B, C, on a :

$$\begin{aligned} A \subseteq B &\Rightarrow A - C \subseteq B - C \\ A \subseteq B &\Rightarrow C - A \subseteq C - B \end{aligned}$$

Un terme τ sera dit *syntaxiquement croissant* (respectivement *décroissant*) par rapport à la variable X si et seulement si, dans l'arbre de syntaxe[2], tout chemin allant de la racine de τ à un feuille étiquetée X, passe par un nombre pair (respectivement impair) de fois par le second argument de l'opérateur de différence ensembliste.

[2] L'arbre de syntaxe d'une expression est un arbre dont les noeuds sont des opérateurs et les feuilles des noms de variables ou des constantes. Par exemple, l'arbre de syntaxe de l'expression $A \cap \texttt{rsrc}(B)$ a le noeud $\cap$ comme racine. Ce noeud a la feuille A comme premier fils et le noeud `rsrc` comme second fils. Le noeud `rsrc` a la feuille B comme unique fils.

Le résultat de cette définition est que, si le terme τ est syntaxiquement croissant (respectivement décroissant) par rapport à la variable X, τ s'interprète comme une fonction monotone (respectivement anti-monotone) de X.

Exemples

$$\tau = \texttt{src}(\texttt{rtgt}(X \cup Y) \cap Z)$$

est un terme si X et Y sont des marques d'états et Z est une marque de transitions. Il est syntaxiquement croissant par rapport à X, Y et Z.

La fonction de $X : X \rightarrow \texttt{src}(\texttt{rtgt}(X \cup Y) \cap Z)$ est donc monotone.

$$\tau = \texttt{src}(\texttt{rtgt}(X - Y) - Z)$$

est un terme syntaxiquement croissant par rapport à X et est un terme syntaxiquement décroissant par rapport à Y et Z.

La fonction de $Y : Y \rightarrow \texttt{src}(\texttt{rtgt}(X - Y) - Z)$ est donc anti-monotone.

2.3.2 Système d'équations aux points fixes

On appelle système d'équations aux points fixes un ensemble fini d'équations :

$$\Sigma = \begin{cases} X_1 = \tau_1 & (\text{de signe } S_1) \\ \vdots & \\ X_p = \tau_p & (\text{de signe } S_P) \end{cases}$$

où

- $X_1, \ldots, X_p$ sont les variables du système d'équations ($p \geq 1$);
- $\tau_1, \ldots, \tau_p$ sont les termes du système d'équations;
- $S_1, \ldots, S_p$ sont les signes (+ ou -) associés aux variables.

Des paramètres $Y_1, \ldots, Y_q$ ($q \geq 1$) sont également définis sur le système d'équations. Les paramètres sont des ensembles constants d'états ou de transitions ; ils peuvent apparaître dans les termes.

Le signe des variables est défini par l'utilisateur lors de la définition d'un opérateur. Ce signe est utilisé pour déterminer la valeur initiale d'une variable et la méthode de calcul de l'équation associée à une variable.

Exemple

Ci-dessous se trouve la définition MEC d'un nouvel opérateur appelé `inevitable`.

```
function inevitable(X : state; Y : trans) return Z:state;
var T:_trans
begin
  Z = X \/ (* - src(T));
  T = Y /\ rtgt(* - Z)
end.
```

Les paramètres sont `X`, de type "états", et `Y`, de type "transitions".

Les termes sont `X \/ (* - src(T))` et `Y /\ rtgt(*- Z)`.

Le résultat de la fonction est la variable `Z`, de type 'états' (`return Z:state`) et de signe '`+`' (par défaut).

La variable `T` est de type 'transitions' (`var T:_trans`). Le caractère "`_`" devant `trans` indique une variable de signe '`-`'.

Remarque

La fonction `inevitable`, définie ci-dessus, retourne l'ensemble des états Z tels qu'il existe un et un seul chemin allant de chaque état z de Z à un état de X, en utilisant les transitions de Y. Elle permet de rechercher, par exemple, tous les états menant obligatoirement à un état bloqué[3] :

```
bloques := * - src(*);
bloquants := inevitable(bloques, *);
```

Propriétés

Les algorithmes de calcul de la plus petite solution d'un système d'équations sont basés sur les propriétés suivantes. Elles justifient notamment les conditions initiales de ces algorithmes.

Plus petite solution Soit $(P_1, \ldots, P_p)$ la plus petite solution de Σ, alors $(P_1, \ldots, P_p)$ est également la plus petite solution de

$$\Sigma' = \begin{cases} X_i = X_i \cup \tau_i & \text{si } S_i = + \\ X_i = X_i \cap \tau_i & \text{si } S_i = - \end{cases} \quad \text{pour tout } 1 \leq i \leq p$$

[3] La définition d'un nouvel opérateur MEC à partir d'une spécification n'est pas triviale. Le lecteur intéressé trouvera des exemples de construction de nouveaux opérateurs MEC dans [1].

Valeur initiale Soit un système de transitions $A = \langle S, T, \alpha, \beta, \lambda \rangle$, S_i le signe de la variable X_i et τ_i le terme associé à la variable X_i. La valeur initiale de la variable X_i est définie comme suit :

$$\begin{cases} X_i^0 = \{\} & \text{si } S_i = + \\ X_i^0 = S & \text{si } S_i = - \text{ et } \tau_i \text{ est de type “états”} \\ X_i^0 = T & \text{si } S_i = - \text{ et } \tau_i \text{ est de type “transitions”} \end{cases}$$

Résolution On définit une suite $(X_1^k, \ldots, X_p^k)$, k étant entier ≥ 0, avec $(X_1^0, \ldots, X_p^0)$ comme valeur initiale et :

$$X_i^{k+1} = \begin{cases} X_i^k \cup \tau_i(X_1^k, \ldots, X_n^k) & \text{si } S_i = + \\ X_i^k \cap \tau_i(X_1^k, \ldots, X_n^k) & \text{si } S_i = - \end{cases} \quad \text{pour tout } 1 \leq i \leq n$$

La plus petite solution de Σ est atteinte lorsque tous les X_i deviennent “stables” :

$$\begin{cases} X_i^{k+1} \subseteq X_i^k & \text{si } S_i = + \\ X_i^{k+1} \supseteq X_i^k & \text{si } S_i = - \end{cases} \quad \text{pour tout } 1 \leq i \leq n$$

Ces propriétés donnent un premier algorithme de calcul de la plus petite solution d'un système d'équations.

2.3.3 Forme réduite des systèmes d'équations

Un système d'équations est sous forme réduite si toutes ses équations ont l'une des formes suivantes :

- $Y = \tau$ où tous les noeuds de τ sont des opérateurs ensemblistes
- $Y = \texttt{src}(Z)$
- $Y = \texttt{tgt}(Z)$
- $Y = \texttt{rsrc}(Z)$
- $Y = \texttt{rtgt}(Z)$

Tout système d'équations peut être transformé en un système d'équations sous forme réduite équivalent. Cette transformation, nécessaire pour l'application de l'algorithme de résolution de MEC (voir Chapitre 3), est automatiquement effectuée par le logiciel lors de la définition d'un nouvel opérateur.

Exemple

Le système d'équations (qui contient une seule équation)

$$\left\{ X = \mathtt{tgt}(T \cap \mathtt{rsrc}(E \cup X)) \right.$$

admet la forme réduite suivante :

$$\begin{cases} X = \mathtt{tgt}(Y1) \\ Y1 = T \cap Y2 \\ Y2 = \mathtt{rsrc}(Y3) \\ Y3 = E \cup X \end{cases}$$

Chapitre 3

Algorithmes MEC

Comme il a été dit dans le Chapitre 1, les deux limitations principales de la version MEC de base sont l'encombrement mémoire et le temps de calcul. Ces limitations sont dues à la représentation explicite des automates synchronisés et à l'explosion combinatoire de la taille des automates.

Dans ce chapitre, nous expliquons plus en détail les structures de données et les algorithmes mis en oeuvre dans cette version de MEC.

3.1 Structures de données

Les structures de données de MEC ont été définies pour optimiser quelques opérations de base très souvent utilisées dans les algorithmes du logiciel. Ces opérations sont:

- retrouver l'état source d'une transition t;
- retrouver l'état cible d'une transition t;
- parcourir toutes les transitions t ayant un état e comme source (*chaînage direct*);
- parcourir toutes les transitions t ayant un état e comme cible (*chaînage inverse*).

Lors de la définition d'un système de transitions (que ce soit manuellement ou lors du calcul du produit synchronisé), MEC numérote les états du système de 0 à NE - 1, NE étant le nombre d'états du système. De même, les transitions sont numérotées de 0 à NT - 1, NT étant le nombre de transitions du système.

Les indices ainsi associés à chaque état et à chaque transition vont permettre un accès direct à plusieurs tableaux que nous décrivons dans les sections sui-

vantes. On appelle cette représentation une *représentation énumérative*, ou *explicite*, parce que chaque état et chaque transition est représenté individuellement.

3.1.1 Représentation énumérative des états et des transitions

Cinq tableaux sont créés pour représenter les états et transitions d'un système de transitions. Trois de ces tableaux ont une taille égale à NT, les deux autres, une taille égale à NE. Nous présentons les tableaux dans l'ordre nécessaire à la réalisation des quatre opérations énumérées page 29. Un exemple complet de représentation d'un automate simple sera donné en Section 3.1.4.

Tableau des états sources Le tableau `origine` associe à chaque transition l'indice de son état source. On a ainsi

$$origine[t] = \text{indice de l'état source de } t$$

L'ordre des éléments de ce tableau n'est pas quelconque: il est trié sur la valeur des indices des états. Les premiers éléments du tableau ont donc la valeur 0, puis les suivants la valeur 1, etc. Cet ordre particulier rend plus aisé le parcours des transitions ayant un état donné comme état source.

Exemple

transitions	`origine`
0	0
1	0
2	1
3	1
4	2
5	2
6	2

Les transitions 0 et 1 ont l'état '0' comme origine; les transitions 2 et 3 ont l'état '1' comme origine; les transitions 4, 5 et 6 ont l'état '2' comme origine.

Tableau des états cibles Un deuxième tableau, appelé `but`, associe à chaque transition l'indice de son état cible. On a ainsi:

$$but[t] = \text{indice de l'état cible de } t$$

Chaînage direct Le *chaînage direct* consiste à accéder à toutes les transitions ayant un état donné comme état source.

Le tableau `prem_trans` associe à chaque état l'indice de la première transition ayant cet état comme source. `prem_trans` est donc utilisé pour accéder au tableau `origine`. La taille du tableau est *NE*. Si un état n'est la source d'aucune transition, la valeur de `prem_trans` associée est négative.

L'algorithme "C" de parcours de toutes les transitions ayant l'état 'e' comme source est le suivant:

```
ref = prem_trans[e];
if ( ref >= 0 )
{
  while ( ref < NT && origine[ref] == origine[ref+1] )
    ref++;
}
```

La variable `ref` va successivement prendre comme valeur les indices des transitions ayant 'e' comme état source. On a donc l'invariant

$$ref \geq 0 \wedge ref < NT \Rightarrow origine[ref] = e$$

Chaînage inverse Le *chaînage inverse* consiste à accéder à toutes les transitions ayant un état donné comme état cible. Le chaînage inverse est rendu possible grâce aux deux tableaux `inv_e` et `inv_trans`.

Le tableau `inv_e` associe à chaque état l'indice de la première transition ayant cet état comme cible. La taille de `inv_e` est *NE*. `inv_e` est utilisé pour accéder au tableau `inv_trans`.

Une entrée de `inv_trans` contient l'indice de la transition suivante ayant le même état cible que l'entrée courante. Un entrée à valeur négative indique la fin de la liste des transitions. La taille de `inv_trans` est *NT*.

L'algorithme "C" de parcours de toutes les transitions ayant l'état 'e' comme cible est le suivant:

```
ref = inv_e[e];
while ( ref >= 0 )
   ref = inv_trans[ref];
```

La variable `ref` va successivement prendre comme valeur les indices des transitions ayant 'e' comme état cible. On a donc l'invariant

$$ref \geq 0 \Rightarrow but[ref] = e$$

3.1.2 Représentation des marques

Une marque est un sous-ensemble des états ou des transitions d'un système de transitions. Une marque a un nom et a pour rôle, le plus souvent, de stocker le résultat du calcul d'une propriété.

Afin d'économiser au maximum l'encombrement mémoire, une marque est codée sous forme d'un tableau de bits. La taille du tableau est soit *NE*, dans le cas d'une marque d'états, soit *NT*, dans le cas d'une marque de transitions. Un bit i d'une marque indique la présence de l'état (de la transition) d'indice i du système de transitions dans la marque. Si le bit est à 0, l'état (la transition) n'est pas présent, s'il est à 1, l'état (la transition) est présent.

3.1.3 Représentation des compteurs

Un compteur associe à chaque état (ou transition) d'un système de transitions une valeur entière. Les compteurs sont utilisé par certains algorithmes de MEC. Par exemple, l'algorithme de parcours en profondeur d'abord (voir Section 3.2.3) numérote les états dans l'ordre de parcours.

Un compteur d'états est un tableau de taille *NE*. Chaque entrée contient la valeur associée à l'état d'indice égal à l'indice de l'entrée. De même, un compteur de transitions a une taille *NT*.

3.1.4 Représentation des automates simples

Un automate simple (par opposition à automate synchronisé) contient un ensemble d'états simples et un ensemble de transitions simples, ainsi qu'un nombre quelconque de marques et de compteurs.

Etats et transitions Les états et transitions d'un automate simple sont représentés à l'aide des cinq tableaux décrits dans la Section 3.1.1.

En plus de ces cinq tableaux, il existe un tableau contenant le nom de chaque état de l'automate et un tableau contenant le label de chaque transition. Ces tableaux sont utilisés pour retrouver les noms des états ou les labels des transitions à partir de leur indice (pour des opérations d'affichage, par exemple).

Exemple

Etant donné la représentation graphique de l'automate simple de la Figure 3.1 et la définition MEC de cet automate (Figure 3.2), les deux tableaux de représentation des noms d'états et des labels de transitions se trouvent

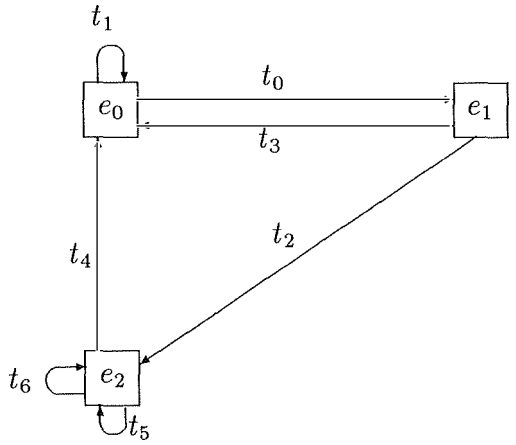


Figure 3.1: Exemple d'automate simple à représenter

```
transition_system auto_simple;
e0 |- t0   -> e1,
      t1   -> e0;
e1 |- t2   -> e2,
      t3   -> e0;
e2 |- t4   -> e0,
      t5   -> e2,
      t6   -> e2;
```

Figure 3.2: Définition MEC de l'automate simple

en Figure 3.3. Les cinq tableaux des états et transitions se trouvent en Figure 3.4 et en Figure 3.5. Pour faciliter la lecture, les indices des états sont en fonte normale et les indices des transitions sont en fonte *italique*.

Marques et compteurs Un nombre quelconque (fini) de marques et de compteurs peut être défini sur un système de transitions. Chaque système de transitions contient donc quatre tableaux supplémentaires: deux tableaux de marques pour les marques d'états et de transitions, et deux tableaux pour les compteurs d'états et de transitions.

3.1.5 Représentation des automates synchronisés

Un automate synchronisé contient un ensemble de multi-états et un ensemble de multi-transitions, ainsi qu'un nombre quelconque de marques et de compteurs.

états	noms
0	e_0
1	e_1
2	e_2

transitions	labels
0	t_0
1	t_1
2	t_2
3	t_3
4	t_4
5	t_5
6	t_6

Figure 3.3: Représentation énumérative de l'automate simple (1)

états	prem_trans
0	*0*
1	*2*
2	*4*

transitions	origine	but
0	0	1
1	0	0
2	1	2
3	1	0
4	2	0
5	2	2
6	2	2

Figure 3.4: Représentation énumérative de l'automate simple (2)

états	inv_e
0	*0*
1	*1*
2	*2*

transitions	inv_trans
0	*-1*
1	*3*
2	*5*
3	*4*
4	*-1*
5	*6*
6	*-1*

Figure 3.5: Représentation énumérative de l'automate simple (3)

Multi-états et multi-transitions Pour rappel, un multi-état est un n-uplet composé d'états des automates composants. Un multi-état composé des indice d'état $e_1, \ldots, e_n$ est noté:

$$e(e_1, \ldots, e_n)$$

De même, une multi-transition est un n-uplet composé de transitions des automates composants. Une multi-transition composé des indices de transition $t_1, \ldots, t_n$ est notée:

$$(t_1, \ldots, t_n)$$

Lors du calcul du produit synchronisé accessible, MEC associe à chaque multi-état et multi-transition un indice unique. La représentation d'un automate synchronisé utilise donc les cinq tableaux de la Section 3.1.1. Ces informations ne sont cependant pas suffisantes, car il est nécessaire de faire un lien entre l'indice d'un multi-état (une multi-transition) et les indices des états (transitions) composants.

Les concepteurs de MEC ont défini un table de *hachage*, qui associe à une suite d'états (de transitions) composants, l'indice du multi-état (de la multi-transition).

$$\begin{array}{rcl} e(e_1, \ldots, e_n) & \longrightarrow & \text{indice multi-état} \\ (t_1, \ldots, t_n) & \longrightarrow & \text{indice multi-transition} \end{array}$$

La clé de hachage est la somme des indices des e_i (ou t_i). Le tableau de hachage est surtout utilisé lors du calcul du produit synchronisé (Section 3.2.1).

L'opération inverse, à savoir associer un indice de multi-état (multi-transition) aux indices des états (transitions) composants, doit également être supportée[1].

3.2 Algorithmes

Dans cette section, nous décrivons brièvement les algorithmes mis en oeuvre dans la version de base de MEC. Le but n'est pas de décrire chaque algorithme en détail, mais plutôt d'expliquer de manière intuitive leur fonctionnement. Cette section est basée sur [4], pages 46 à 102 et sur [1].

[1] Nous n'entrons pas dans les détails, mais d'autres tableaux sont définis pour pouvoir réaliser cette opération.

Etats à construire = $\emptyset$
Empiler les états initiaux sur *Etats à construire* **(1)**
Tant que *Etats à construire* $\neq \emptyset$
  état courant:= dépiler(*Etats à construire*) **(2)**
  Ajouter_état(*état courant*) **(3)**
  Ajouter_transitions(*état courant*) **(4)**
fin tant que

Algorithme 3.1: Calcul du produit synchronisé (version MEC)

3.2.1 Calcul du produit synchronisé accessible

Pour le calcul du produit synchronisé (voir Section 2.1.2), on dispose d'une contrainte de synchronisation et d'une liste d'automates composants. Le résultat du calcul est un automate synchronisé répondant à la définition du produit synchronisé accessible.

L'algorithme de calcul du produit synchronisé se trouve en Figure 'Algorithme 3.1'. Le principe de l'algorithme est le suivant: on considère deux ensembles d'états, à savoir les états à construire et les états construits. L'ensemble des états à construire est constitué des multi-états dont on doit encore calculer les transitions applicables (successeurs). L'ensemble des états construits constitue l'ensemble des états de l'automate synchronisé auxquels on a déjà appliqué toutes les transitions de la contrainte de synchronisation. A chaque itération de l'algorithme, on retire un état de l'ensemble des états à construire, on l'ajoute dans l'ensemble des états construits et on essaie d'appliquer toutes les transitions globales à cet état. A chaque fois qu'une transition est applicable, on ajoute la transition dans l'ensemble des transitions de l'automate synchronisé et on ajoute l'état but de la transition dans l'ensemble des états à construire, si cet état but n'a pas encore été traité.

- L'instruction **(1)** consiste à empiler tous les états initiaux de l'automate synchronisé dans l'ensemble *Etats à construire.* Pour rappel, l'ensemble des états initiaux de l'automate synchronisé est le produit cartésien des états initiaux des automates composants.
- L'instruction **(2)** consiste à retirer un état de l'ensemble *Etats à construire* et d'assigner cet état à *état courant. état courant* est donc un multi-état.
- L'instruction **(3)** ajoute *état courant* dans l'ensemble des états de l'automate synchronisé.
- La procédure "Ajouter_transitions" **(4)** ajoute toutes les transitions applicables à *état courant.* On entend par "transitions applicables", l'ensemble

```
origine(état e)
Début
   S'il existe une transition issue de e et
   Si cette transition est marquée argument
      alors affecter e à la marque affecté
      sinon retirer e de la marque affecté
Fin
```

Algorithme 3.2: Calcul de l'opérateur élémentaire `src` (procédure origine)

des multi-transitions qui respectent la contrainte de synchronisation et telles que chaque sous-transition à pour état source un sous-état de l'automate composant correspondant.

Chaque fois qu'une transition applicable est trouvée, la procédure ajoute la transition dans l'automate synchronisé. Si l'état but de la transition n'est présent ni dans l'ensemble *Etats à construire*, ni dans l'ensemble des états déjà construits, il est ajouté dans *Etats à construire*.

Cette dernière opération étant répétée un très grand nombre de fois lors du calcul, l'importance du rôle de la table de hachage (voir Section 3.1.5) s'explique. En effet, grâce à cette table, il est possible de détecter très rapidement si l'état but d'une transition est déjà présent ou pas dans l'automate.

3.2.2 Calcul des opérateurs élémentaires

L'évaluation des opérateurs élémentaires se fait en parcourant la table des états et en appliquant à chacun la procédure correspondant au calcul à effectuer. La marque argument de la fonction est désignée par *argument*, la marque *affecté* correspond à la marque résultat de la fonction.

Par exemple, le principe de la procédure "origine", qui est est appelée pour le calcul de `src`(T), se trouve en Figure 'Algorithme 3.2'. La marque *argument* est un marque de transitions contenant les transitions de T. Lorsque la procédure "origine" a été appelée pour tous les états du système, la marque *affecte* est affectée à l'ensemble des états du système de transitions qui sont la source d'au moins une transition marquée T.

A titre d'illustration de l'utilisation de structures de données vues en Section 3.1, la Figure 'Algorithme 3.3' contient une implémentation possible (en "C") de la procédure "origine'.

Les autres opérateurs élémentaires sont évalués au moyen de procédures du même type que "origine". Pour l'évaluation de `rsrc` et `rtgt` la procédure associée parcourt toutes les transitions partant de (ou arrivant à) l'état passé en argument (utilisation du chaînage direct ou inverse).

```
void origine(int e, marque argument, marque affecte)
{
  /* Tester bit ´prem_trans[e]´ de argument */
  if (prem_trans[e] >= 0 && tester_bit(argument, prem_trans[e]))
    mettre_bit_a_1(affecte, e);  /* bit ´e´ de affecte à 1 */
  else
    mettre_bit_a_0(affecte, e);  /* bit ´e´ de affecte à 0 */
}
```

Algorithme 3.3: Implémentation "C" de la procédure origine

Retenons simplement que l'évaluation des quatre opérateurs élémentaires est linéaire sur le nombre d'états (ou transitions) du système de transitions.

3.2.3 Calcul des composantes fortement connexes

L'algorithme de calcul des composantes fortement connexe est une variante de l'algorithme de Tarjan. Un système de transitions est considéré comme un graphe orienté dont les sommets sont les états du système et les arcs, les transitions.

MEC contient un algorithme générique de parcours en profondeur d'abord (*DFS* ou *Depth First Search*) du graphe correspondant à un système de transitions. L'algorithme DFS visite chaque état du système et chaque transition issue de l'état en cours de visite (chaînage direct). Lorsqu'un état est visité, il est marqué comme tel dans une marque *visité*. Pour chaque transition de l'état en cours de visite, l'état but de la transition est visité par un appel récursif, sauf s'il est déjà marqué *visité*. Cet algorithme est donc linéaire sur le nombre d'états et de transitions du système de transitions.

Dans le cadre du calcul de `loop`, l'algorithme de parcours DFS est utilisé et deux compteurs d'états sont créés. Le compteur *numéro* associé à chaque état le numéro de l'état dans l'ordre du parcours DFS. Le compteur *lowlink* associe à chaque état le plus petit numéro d'état auquel on a pu accéder lors du parcours. Grâce à ces deux compteurs, à une pile d'états et à deux marques supplémentaires sur les transitions, il est possible de déterminer les composantes fortement connexes du graphe.

Nous ne détaillons pas plus cet algorithme, car il est assez complexe à décrire. Le lecteur intéressé pourra se référer à [1] et [4] pour une explication complète. Notre but ici est de montrer que le calcul de `loop` est linéaire sur le nombre d'états et de transitions du système et qu'il requiert l'utilisation de compteurs d'états.

3.2.4 Calcul des états accessibles et co-accessibles

Le calcul des états accessibles (opérateur `reach`) utilise le parcours DFS. Lors de la visite d'un état, l'état ne sera marqué *visité* que s'il fait partie des états en arguments de `reach`. Le résultat est l'ensemble des états marques *visité*.

De nouveau, cet algorithme est linéaire sur le nombre d'états/transitions du système.

Le calcul de `coreach` est plus complexe, car il faut utiliser le chaînage inverse. L'implémentation utilise cependant le parcours DFS et est une variante de l'algorithme de calcul de `loop`. Il requiert donc des compteurs et des marques d'états et de transitions, et il est linéaire sur la taille de l'automate.

3.2.5 Calcul des plus courts chemins

L'algorithme de calcul de `trace`$(Q1, R, Q2)$ doit retourner un ensemble de transitions appartenant à R formant un plus court chemin entre les états de $Q1$ et de $Q2$. Le principe de l'algorithme est de calculer les états co-accessibles en une transition à partir de $Q2$, puis en en deux transitions et ainsi de suite jusqu'à ce qu'au moins un état co-accessible soit inclus dans $Q1$.

Au point de vue complexité, l'algorithme a donc les mêmes caractéristiques que `coreach`.

3.2.6 Calcul des points fixes d'un système d'équations

L'algorithme du calcul des points fixes d'un système d'équations est linéaire sur le nombre d'états et de transitions du système de transitions. Une marque d'états (de transitions) est associée à chaque équation de type "état" ("transition") du système. Cette marque contient l'ensemble des états (transitions) résultant de l'évaluation. Leur valeur initiale dépend du signe de la variable de chaque équation (voir Section 2.3.2). Une marque associée à une variable de signe + est initialisée à "vide". Une marque associée à un variable de signe − est initialisée à "ensemble de tous les états" ou de "toutes les transitions".

L'idée de l'algorithme est d'effectuer un parcours DFS de chaque état et chaque transition du système. A chaque fois qu'un état est visité, l'algorithme cherche à déterminer le plus rapidement possible la valeur définitive des marques d'états du système et à répercuter les changements dans toutes les équations (et uniquement dans les équations où il est nécessaire de le faire). Il en est de même pour les transitions. Pour fonctionner selon ce principe, l'algorithme a besoin d'un système d'équations sous forme réduite, de marques d'états et de transitions et d'un compteur d'états.

De nouveau, nous n'entrons pas plus en détail dans l'explication, car elle est assez complexe (voir [1] et [4]). Cependant, on peut remarquer que l'algorithme est linéaire sur le nombre d'états et de transitions du système, et qu'il nécessite également un compteur d'états.

Le but de ce chapitre était de présenter, sans entrer dans trop de détails, le fonctionnement interne de la version MEC de base. Le but était surtout de comprendre pourquoi, dans le chapitre suivant où nous introduisons les Arbres Partagés, il a été nécessaire de créer un nouvel algorithme dédié aux Arbres Partagés pour chaque opérateur du logiciel.

Chapitre 4

Intégration des Arbres Partagés dans MEC

Dans le chapitre précédent, nous avons expliqué le fonctionnement interne de la version MEC de base. Les états et transitions d'un automate sont représentés explicitement dans des tableaux et tous les algorithmes sont linéaires sur la taille de ces tableaux.

Cette représentation explicite a l'avantage d'être intuitive, mais limite fortement le champ des problèmes traitables par le logiciel. En effet, comme nous l'avons dit au Chapitre 1, la taille des automates synchronisés a tendance à croître exponentiellement avec le nombre d'automates composants (voir Chapitre 6 pour des exemples). Des efforts ont été faits par les concepteurs du logiciel pour contourner ces problèmes (par exemple, les marques sont codées sous forme de tableaux de bits), mais cela reste insuffisant.

En pratique, sur une station de travail moyenne, le logiciel est limité à des automates synchronisés comportant une centaine de millier d'états et de transitions. Sur une grosse station de travail, il est possible de monter à quelques centaines de milliers.

Dans la première partie de ce chapitre (Section 4.1), nous présentons une structure de données permettant, dans beaucoup de cas, de représenter sous forme très compacte un ensemble de n-uplets. Cette structure de données est appelée *Arbre Partagé*[1]. Toutes les opérations ensemblistes (union, intersection, ...) et relationnelles (projection, jointure, ...) existent dans une bibliothèque (voir [11]). Ces algorithmes travaillent de manière globale sur la structure, et non pas itérativement sur les éléments représentés.

Ensuite (Section 4.2), nous expliquons comment les "objets" MEC (systèmes

[1] *Sharing Tree* en anglais

de transitions synchronisés, marques et compteurs) sont représentés à l'aide des Arbres Partagés.

La troisième partie du chapitre (Section 4.3) est consacrée à l'étude des nouveaux algorithmes dédiés aux Arbres Partagés implémentant les opérateurs du logiciel MEC vus au Chapitre 2. A chaque opérateur MEC correspond un algorithme dédié, travaillant de manière globale sur les ensembles, et non plus de manière itérative, comme les algorithmes présentés au Chapitre 3. Tous les algorithmes présentés dans cette section sont nouveaux par rapport à ceux existant dans la bibliothèque de base, excepté la première version du calcul du produit accessible (voir Section 4.3.2).

Dans la dernière partie du chapitre (Section 4.4), nous expliquons deux améliorations dans la gestion des Arbres Partagés. Ces améliorations permettent d'augmenter la vitesse de calcul des algorithmes dédiés. Nous avons implémenté ces améliorations après que certaines expérimentations ont montré des faiblesses dans la bibliothèque existante de gestion des Arbres Partagés.

4.1 Les Arbres Partagés

Dans cette section, nous introduisons la définition formelle des Arbres Partagés et les caractéristiques générales des algorithmes de manipulation des Arbres Partagés(voir [11]).

4.1.1 Définition

Un *Arbre Partagé* est un graphe sans cycle possédant une racine. On définit formellement un Arbre Partagé par le quadruplet $(N, V, val, succ)$ où

- $N = N_0 + \ldots + N_k, k \geq 0$ est l'ensemble (fini) des noeuds (les noeuds sont rangés par couche, N_i est l'ensemble des noeuds de la couche i) ;
- $val : N \rightarrow V + \{\top, \bot\}$ est la fonction qui associe une valeur à un noeud ;
- $succ : N \nrightarrow \wp(N)$ donne l'ensemble des successeurs d'un noeud.

et les règles suivantes sont respectées :

1. $\forall 0 \leq i < k, \forall n \in N_i, succ(n) \subseteq N_{i+1}$: chaque noeud a tous ses successeurs dans la couche suivante ;
2. $\forall 0 \leq i \leq k, \forall n_1 \ \& \ n_2 \in N_i, n_1 \neq n_2, val(n_1) = val(n_2) \Rightarrow succ(n_1) \neq succ(n_2)$: deux noeuds de même valeur dans la même couche n'ont pas les mêmes fils (canonicité) ;

3. $\forall n \in N, \forall s_1 \ \& \ s_2 \in succ(n), s_1 \neq s_2 \Rightarrow val(s_1) \neq val(s_2)$: un noeud n'a pas de fils de même valeur ;

4. $\#N_0 = 1$ et $\forall n \in N, val(n) = \top \Leftrightarrow n \in N_0$: la première couche N_0 contient un seul élément (appelé la *racine*), le seul ayant la valeur $\top$;

5. $val(n) = \bot \Rightarrow succ(n) = \emptyset$ et $succ(n) = \emptyset \Rightarrow (val(n) = \bot \vee val(n) = \top)$: un noeud de valeur $\bot$ n'a pas de successeur et un noeud sans successeur a soit la valeur $\bot$, soit la valeur $\top$.

On définit l'ensemble des éléments d'un Arbre Partagé $AP = (N, V, val, succ)$ comme l'ensemble des n-uplets présents sur tous les chemins partant de la racine $r \in N_0$. Formellement, on a :

$\mathcal{E}lem(AP) = Set(r)$, et $\forall n \in N$:

$$Set(n) = \begin{cases} \{()\} & \text{si } val(n) = \bot \\ \bigcup_{s \in succ(n)} Set(s) & \text{si } val(n) = \top \\ \{val(n)\} \times \bigcup_{s \in succ(n)} Set(s) & \text{sinon} \end{cases}$$

La structure des Arbres Partagés, telle qu'elle a été définie, assure le partage maximal des préfixes des n-uplets, et le partage des suffixes lorsque c'est possible.

Une autre propriété intéressante est l'isomorphisme de la structure par rapport aux éléments représentés. Ainsi, pour un ensemble de n-uplets donné, il existe une et une seule structure d'Arbre Partagé correspondant à l'ensemble, et vice-versa.

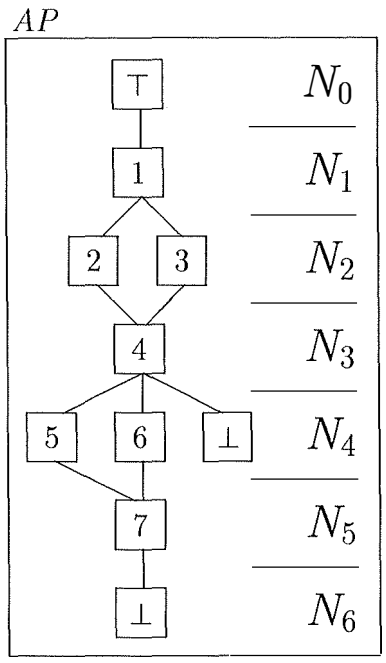
Exemple

La Figure 4.1 représente un Arbre Partagé "AP" et les éléments qu'il contient. $N_0, \ldots, N_6$ sont les couches de AP.

4.1.2 Principes de base des algorithmes

Dans cette section, nous décrivons les principes de base des algorithmes de construction d'un Arbre Partagé. Le but de montrer comment il est possible de travailler de manière globale sur un Arbre Partagéen respectant la canonicité de l'arbre. Le lecteur intéressé pourra se référer à [11] et [10] pour une description détaillée de tous les algorithmes ensemblistes et relationnels sur les Arbres Partagés.

On entend par "travailler de manière globale sur un Arbre Partagé", le fait que les algorithmes ne manipulent pas itérativement tous les éléments représentés, mais parcourent les couches, noeuds et successeurs récursivement de telle sorte



$$\mathcal{E}lem(AP) = \{\ (1,2,4,5,7),\ (1,2,4),\ (1,2,4,6,7),\ (1,3,4,5,7),\ (1,3,4),\ (1,3,4,6,7)\ \}$$

Figure 4.1: Exemple d'arbre partagé

que le résultat d'une opération soit un Arbre Partagé représentant l'ensemble des éléments répondant à la définition de l'opération effectuée.

En général, pour effectuer une opération globale sur un ou plusieurs Arbres Partagés (union, intersection, ...), les algorithmes ne modifient jamais directement la structure interne d'un arbre existant, mais créent un nouvel arbre. Par exemple, pour faire l'union de deux arbres, l'algorithme d'union prend deux arbres en entrée et crée en sortie un arbre représentant l'union des deux arbres.

Le principe de fonctionnement des algorithmes est un parcours récursif de haut en bas (*top-down*) de tous les noeuds et successeurs des Arbres Partagés en entrée et une construction de bas en haut (*bottom-up*) de l'Arbre Partagé résultat. La construction bas en haut assure la préservation de la forme canonique d'un arbre (voir règle 2 de la définition d'un Arbre Partagé, Section 4.1.1).

Pour illustrer ces principes, nous considérons l'algorithme d'union de deux Arbres Partagés. Soit les trois Arbres Partagés AP_1, AP_2 et AP_U représentés en Figure 4.2, où AP_U est l'union de AP_1 et AP_2. En entrée, l'algorithme d'union reçoit AP_1 et AP_2 et doit générer AP_U en sortie.

L'algorithme commence par parcourir les successeurs du couple de noeuds $\langle \top_{AP_1}, \top_{AP_2} \rangle$. Dans un premier temps, le noeud $\top_{AP_U}$ est créé (sans être ajouté dans la couche, ce sera fait plus tard), et un appel récursif à l'algorithme d'union est effectué pour le couple de successeurs $\langle 1_{AP_1}, 1_{AP_2} \rangle$. Le résultat de cet appel

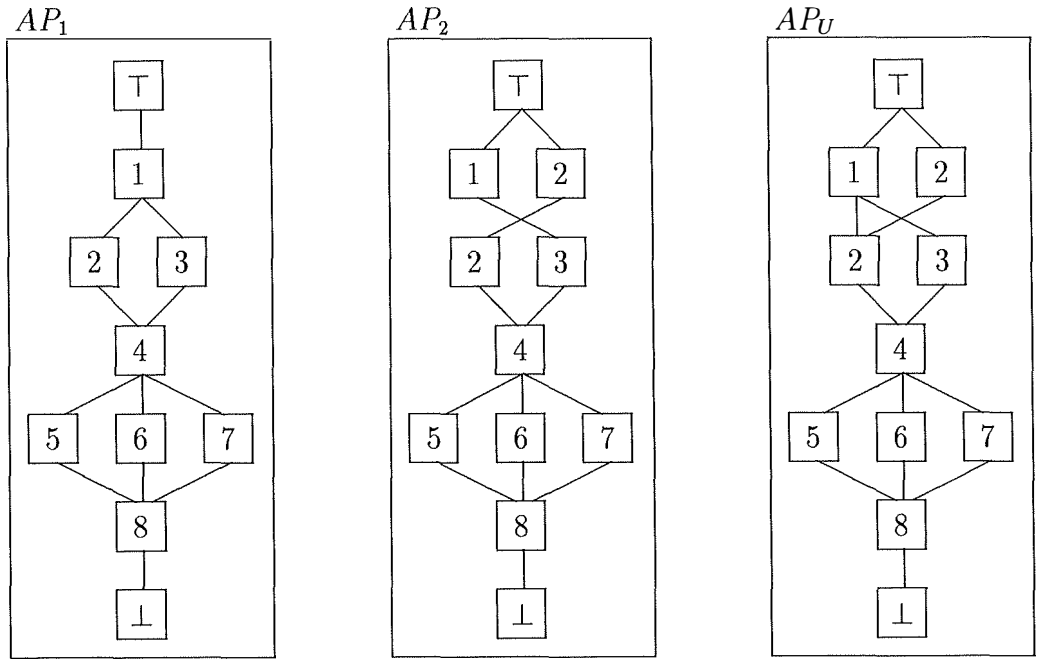


Figure 4.2: Exemple d'union de deux Arbres Partagés

est le sous-arbre de racine 1_{AP_U}. Il a déjà été ajouté dans la couche 1 de AP_U, et il reste à l'ajouter dans l'ensemble des successeurs de $\top_{AP_U}$. La Figure 4.3 représente la structure de AP_U à ce moment de l'algorithme.

L'algorithme peut ensuite continuer à parcourir les successeurs de $\langle \top_{AP_1}, \top_{AP_2} \rangle$. Le seul successeur de $\top_{AP_1}$ a déjà été parcouru, mais il reste le noeud 2_{AP_2}. L'algorithme appelle alors l'algorithme de création d'une copie d'un sous-arbre, avec le noeud 2_{AP_2} comme argument. Au retour de cet algorithme, le sous-arbre de racine 2_{AP_U} a été créé. Le noeud 2_{AP_U} est alors ajouté comme successeur de $\top_{AP_U}$.

Le deux ensembles des successeurs de $\langle \top_{AP_1}, \top_{AP_2} \rangle$ ayant été entièrement parcourus, le noeud $\top_{AP_U}$ est inséré dans la couche 0 de AP_U. La structure de AP_U est alors définitive.

Quelques remarques supplémentaires sont encore nécessaires pour bien comprendre le mécanisme de construction de AP_U :

- La condition d'arrêt des appels récursifs est la rencontre des noeuds $\langle \bot_{AP_1}, \bot_{AP_2} \rangle$. Ces noeuds n'ayant jamais de successeurs, aucun appel récursif n'est plus effectué.

- Les noeuds sont ajoutés dans les couches *après* que les appels récursifs sur tous les successeurs sont terminés, ce qui implique bien une construction

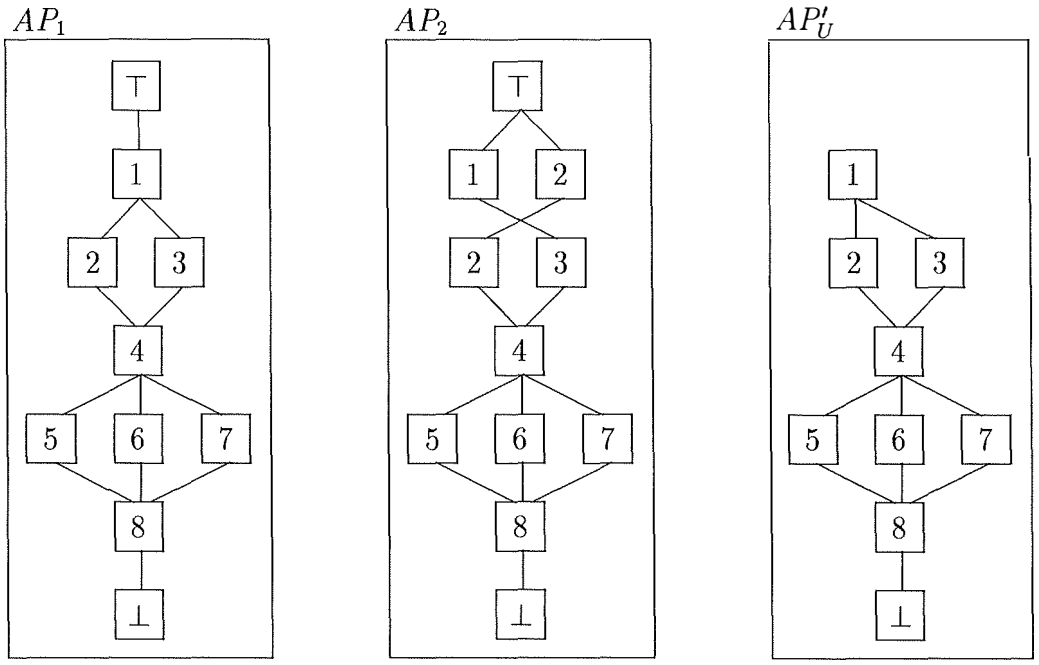


Figure 4.3: Résultat intermédiaire de l'union de deux Arbres Partagés

bottom-up de AP_U.

Le premier noeud à être inséré dans une couche de AP_U est le noeud $\bot_{AP_U}$. Ensuite, le noeud 8_{AP_U} est inséré dans l'avant dernière couche, et ainsi de suite.

- A chaque fois qu'un noeud est inséré dans une couche de AP_U, il faut vérifier qu'il n'existe pas déjà un noeud ayant la même valeur et le même ensemble de successeurs (voir règle 2 de la définition d'un Arbre Partagé). Lorsque cette situation se présente, le noeud "doublon" est détruit et le résultat de l'appel récursif en cours est le noeud existant.

  Ce mécanisme assure la canonicité de AP_U. Par exemple, lorsque l'algorithme de copie du sous-arbre de racine 2_{AP_2} est appelé, tous les noeuds qui sont créés récursivement à partir de la couche 2 existent déjà dans AP_U. D'un point de vue de la structure de AP_U, la copie du sous-arbre 2_{AP_2} se résume donc à la création du noeud 2_{AP_U}, ayant le noeud 2_{AP_U} de la couche 3 comme successeur.

- Tous les algorithmes ensemblistes sont des variantes de l'algorithme d'union. Par exemple, pour un algorithme d'intersection, seuls les successeurs de même valeur font l'objet d'appels récursifs. Ainsi, pour la construction de l'intersection de AP_1 et AP_2, le sous-arbre de racine 2_{AP_2} sera ignoré, puisqu'il n'est pas possible que ses éléments fassent partie du résultat de

l'intersection.

Memoization

L'algorithme d'union qui a été expliqué dans la section précédente fonctionne correctement, mais est proportionnel au nombre d'éléments représentés. En effet, tous les noeuds et tous leurs successeurs sont parcourus récursivement. On parcourt ainsi tous les chemins possibles allant de la racine $\top$ à l'anti-racine $\bot$. Si on examine AP_1 dans la Figure 4.2, il existe 6 chemins différents allant de la racine à l'anti-racine. Ce nombre correspond exactement au nombre d'éléments représentés. De même pour AP_2, il y a 6 chemins et 6 éléments représentés. Or, il est clair que le résultat de l'union des sous-arbres $\langle 4_{AP_1}, 4_{AP_2} \rangle$ donnera toujours le même résultat, à savoir le sous-arbre 4_{AP_U}. De nombreux calculs sont donc redondants.

On peut résoudre ce problème en utilisant une mémoire cache, dans laquelle chaque résultat d'un calcul intermédiaire est stocké. Lorsqu'un nouvel appel récursif doit être effectué, il suffit de consulter cette mémoire cache pour déterminer si un calcul a déjà été effectué. Si c'est le cas, il n'y a pas d'appel récursif et le résultat est immédiat.

La mémoire cache diminue fortement la complexité temporelle des algorithmes, qui deviennent alors linéaires sur le nombre de successeurs, et non plus sur le nombre d'éléments représentés. En général, le nombre de successeurs est beaucoup plus petit que le nombre d'éléments représentés[2]. On trouvera la justification complète de la complexité temporelle des algorithmes en [11].

L'action de stocker des résultats intermédiaires dans une mémoire cache et de les retrouver plus tard est appelée *memoization*. Dans la bibliothèque actuelle des Arbres Partagés (voir [11]), il existe trois types de memoization, qui dépendent du nombre d'arbres en entrée et en sortie.

Memoization dans les noeuds

La memoization dans les noeuds est utilisée par les algorithmes qui reçoivent un Arbre Partagé en entrée et qui génère un Arbre Partagé en sortie. Etant donné que ce type d'algorithme parcourt récursivement tous les noeuds et tous leurs successeurs, il arrive très souvent qu'il existe plusieurs chemins menant au même noeud. Le résultat de chaque opération sur un noeud de l'Arbre Partagé en entrée est donc stocké dans un champ spécial de la structure "noeud".

Si on prend l'exemple de l'algorithme de création d'une copie du sous-arbre 2_{AP_2} en Figure 4.2, il existe trois chemins menant au noeud 8_{AP_2}. Lors du

[2] Ce n'est pas le cas dans les exemples d'Arbre Partagé présentés dans cette section, car le nombre d'éléments est petit, mais c'est souvent le cas lorsque le nombre d'éléments augmente.

premier passage, le sous-arbre 8_{AP_U} est créé. Ce résultat est alors stocké dans un champ spécial du noeud 8_{AP_2}. Lors du passage suivant, ce champ est examiné et le résultat est immédiat.

Memoization dans une table de hachage simple

La table de hachage simple associe à deux noeuds en entrée, un noeud résultat. La table est utilisée, par exemple, dans l'algorithme d'union pour éviter de calculer plusieurs fois l'union de deux sous-arbres ayant des racines identiques.

Dans l'exemple de la Figure 4.2, après le premier parcours des noeuds $\langle 8_{AP_1}, 8_{AP_2} \rangle$, le résultat, c'est-à-dire le sous-arbre 8_{AP_U}, est stocké dans la table de hachage simple :

$$\langle 8_{AP_1}, 8_{AP_2} \rangle \rightarrow 8_{AP_U}$$

Lorsque l'algorithme d'union doit de nouveau calculer l'union des deux sous-arbres (cela se produira exactement trois fois), la table de hachage est consultée et le résultat est directement retourné.

Memoization dans une table de hachage double

La table de hachage double fonctionne selon le même principe que la table de hachage simple, mais au lieu de garder un seul noeud résultat, elle en conserve 2. Cette table est utilisée par un algorithme qui doit générer 2 Arbres Partagés en sortie (voir Section 4.3.2 pour un exemple).

Gestion de la mémoire

Etant donné que les structures des Arbres Partagés sont très dynamiques et qu'on utilise souvent des Arbres Partagés temporaires, il est nécessaire d'éviter le plus possible des appels aux fonctions d'allocation (et libération) de mémoire du système d'exploitation, car leur temps d'exécution n'est pas négligeable.

Pour ce faire, un ensemble de primitives assurent une gestion efficace de la mémoire au moyen de *Booked Memories*. Ces mémoires ont pour rôle de gérer l'allocation et la désallocation d'enregistrements dont la taille est fixe. Les enregistrements libres sont reliés par une liste chaînée.

Les opérations de base sur une mémoire sont l'allocation, qui consiste à retirer le premier élément de la liste, et la désallocation qui consiste à ajouter l'élément à libérer en début de liste. Ces deux opérations sont donc extrêmement rapides (quelques instructions en langage "C").

Lors d'une allocation dans une *booked memory* dont la liste est vide, la primitive d'allocation demande au système d'exploitation l'allocation d'un bloc de mémoire

pouvant contenir un grand nombre d'enregistrements. Les enregistrements ainsi alloués sont ensuite reliés entre eux et ajoutés dans la liste.

Au démarrage du système de gestion de Arbres Partagés, plusieurs de ces mémoires sont créées, pour assurer l'allocation de tous les types d'enregistrements utilisés par le système, à savoir, en autre, une mémoire pour les enregistrements "noeuds", "successeurs", "couches", ...

4.2 Représentation d'automates synchronisés

Nous l'avons vu au Chapitre 2, un système de transitions synchronisé est composé d'un ensemble de multi-états et de multi-transitions (n-uplets). En plus de ces deux ensembles, des marques et des compteurs peuvent être associés au système de transitions. Un marque est soit un sous-ensemble de l'ensemble des multi-états du système, soit un sous-ensemble de l'ensemble des multi-transitions. Le cas des compteurs est un peu différent, puisqu'il s'agit d'associer une valeur entière positive à chaque état ou chaque transition d'un automate.

Dans cette section, nous montrons comment représenter tous ces ensembles avec des Arbres Partagés.

4.2.1 Représentation d'un ensemble de multi-états

MEC identifie chaque état d'un automate par un indice (voir Chapitre 3). Un multi-état d'un automate synchronisé est donc un n-uplet contenant des indices des états des automates composants. Un multi-état composé de n sous-état est donc un n-uplet :

$$e(e_1, \ldots, e_n)$$

où chaque e_i est un indice d'un état du $i^{ème}$ automate composant.

Dans un Arbre Partagé, les multi-états sont représentés "verticalement", c'est-à-dire que chaque couche correspond à un automate composant et que les noeuds d'une couche contiennent uniquement des indices d'états de l'automate correspondant à la couche.

Exemple

Soit l'automate synchronisé de la Figure 4.4. On suppose que les automates composants sont nommés $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$ et $\mathcal{A}_4$. L'ensemble des multi-états à représenter est :

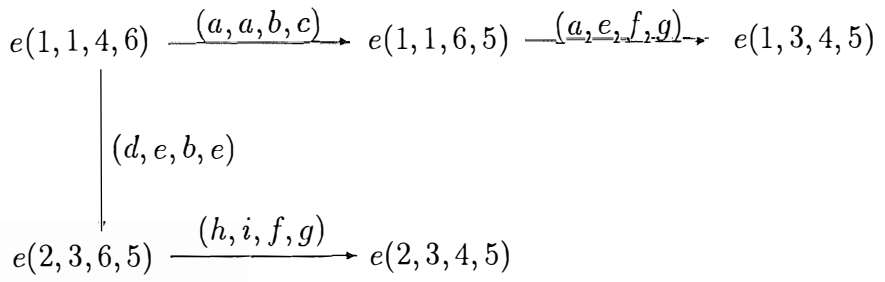
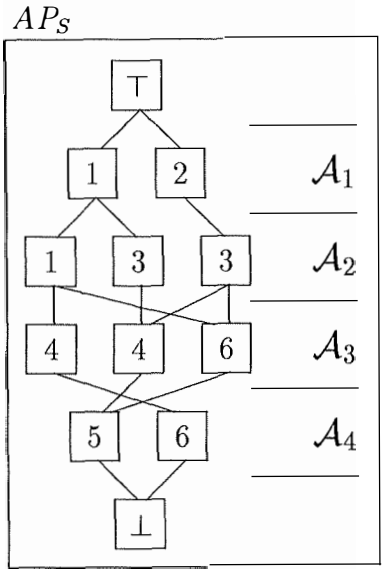


Figure 4.4: Exemple d'automate synchronisé à représenter



$$\mathcal{E}lem(AP_S) = \{\ (1,1,4,6),\ (1,1,6,5),\ (1,3,4,5),\ (2,3,4,5),\ (2,3,6,5)\ \}$$

Figure 4.5: Exemple d'Arbre Partagé multi-états

$$S = \{\ e(1,1,4,6),\ e(1,1,6,5),\ e(1,3,4,5),\ e(2,3,4,5),\ e(2,3,6,5)\ \}$$

La Figure 4.5 représente l'Arbre Partagé AP_S contenant les multi-états de cet automate[3].

[3]Pour simplifier la représentation, les noms des états composants sont affichés dans les noeuds, alors que, en pratique, la valeur contenue dans un noeud est un indice d'état composant. Il y a cependant une correspondance bi-univoque entre un indice d'état et son nom (voir Chapitre 3). Nous continuerons à utiliser cette convention pour faciliter les représentations.

4.2.2 Représentation d'un ensemble de multi-transitions

Le principe de la représentation d'un ensemble de multi-transitions avec un Arbre Partagé est identique à celui vu en section précédente pour les ensemble de multi-états : chaque transition composante d'une multi-transition est stockée dans un noeud, dans la couche correspondant à l'automate composant.

Cependant, comme nous le verrons dans les sections suivantes, l'indice d'une transition n'est pas une valeur utile en soi. La plupart des algorithmes qui manipulent des transitions ont en effet besoin de l'état source et de l'état cible de la transition.

Chaque noeud d'un Arbre Partagé "multi-transitions" contient donc 3 valeurs :

1. l'indice de la transition composante (qui sert d'identifiant du noeud);
2. l'indice de l'état origine de la transition composante;
3. l'indice du l'état but de la transition composante.

Exemple

Soit l'automate synchronisé de la Figure 4.4. L'ensemble des multi-transitions à représenter est :

$$\begin{aligned} T = \{\ & e(1,1,4,6) \xrightarrow{(a,a,b,c)} e(1,1,6,5), \\ & e(1,1,6,5) \xrightarrow{(a,e,f,g)} e(1,3,4,5), \\ & e(1,1,4,6) \xrightarrow{(d,e,b,e)} e(2,3,6,5), \\ & e(2,3,6,5) \xrightarrow{(h,i,f,g)} e(2,3,4,5)\ \} \end{aligned}$$

La Figure 4.6 représente l'Arbre Partagé AP_T contenant les multi-transitions de l'automate. Chaque multi-transition est décomposée en ses transitions composantes et chaque triplet ainsi obtenu devient la valeur d'un noeud de l'Arbre Partagé[4] :

$$\begin{aligned} \mathcal{E}lem(AP_T) = \{\ & (1 \xrightarrow{a} 1,\ 1 \xrightarrow{a} 1,\ 4 \xrightarrow{b} 6,\ 6 \xrightarrow{c} 5), \\ & (1 \xrightarrow{a} 1,\ 1 \xrightarrow{e} 3,\ 6 \xrightarrow{f} 4,\ 5 \xrightarrow{g} 5), \\ & (1 \xrightarrow{d} 2,\ 1 \xrightarrow{e} 3,\ 4 \xrightarrow{b} 6,\ 6 \xrightarrow{e} 5), \\ & (2 \xrightarrow{h} 2,\ 3 \xrightarrow{i} 3,\ 6 \xrightarrow{f} 4,\ 5 \xrightarrow{g} 5)\ \} \end{aligned}$$

[4]Nous représentons la valeur d'un noeud "transition" de label l, de source s et de but b par la notation $s \xrightarrow{l} b$.

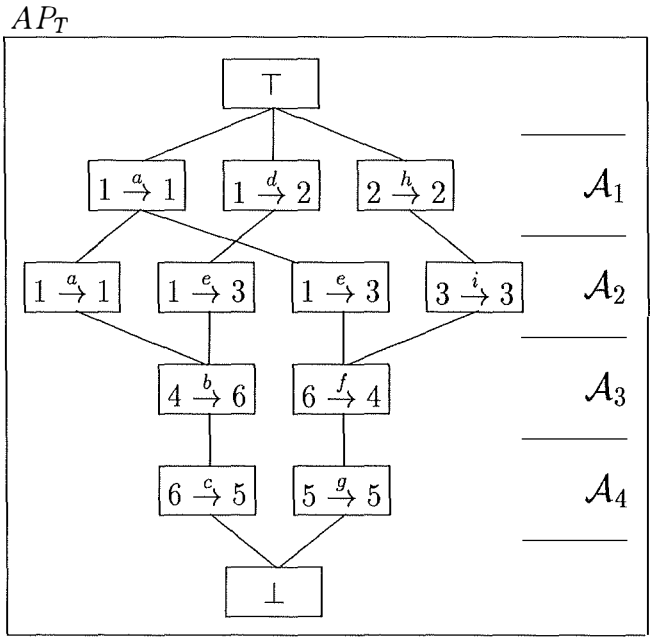


Figure 4.6: Exemple d'Arbre Partagé multi-transitions

4.2.3 Représentation des marques

Une marque est un sous-ensemble des états ou des transitions d'un automate. Chaque marque définie sur un automate synchronisé représenté par Arbres Partagés est donc elle-même un Arbre Partagé. Selon le type de la marque, l'Arbre Partagé est un arbre représentant des multi-états ou des multi-transitions.

4.2.4 Représentation des compteurs

Certains algorithmes de MEC requièrent des compteurs d'états ou de transitions. Un compteur associe à chaque état ou transitions d'un automate une valeur entière positive (codée sur 32 bits).

Chaque compteur d'un automate synchronisé représenté par Arbres Partagés est stocké dans 32 Arbres Partagés, le type des Arbres Partagés dépendant du type du compteur. Un Arbre Partagé i représente l'ensemble des états ou transitions dont le $i^{ème}$ bit de la valeur du compteur est à 1.

Pour déterminer la valeur associée à un état ou à une transition, il faut vérifier, pour chaque Arbre Partagé, si le n-uplet est présent ou pas. Si lors du test, l'élément est présent dans le $i^{ème}$ Arbre Partagé, le $i^{ème}$ bit de la valeur associée à l'élément est à 1, sinon elle est à 0.

Il est évident que cette implémentation est très inefficace, mais les compteurs ont

été implémentés uniquement dans un but de compatibilité avec les algorithmes existants de MEC (voir Chapitre 3). Tous ces algorithmes ont été remplacés par des algorithmes dédiés aux Arbres Partagés qui ne nécessitent pas de compteurs. Les compteurs d'Arbres Partagés ne sont donc plus utilisés dans la version finale du logiciel.

4.3 Algorithmes dédiés aux Arbres Partagés

Comme nous l'avons vu au Chapitre 3, les opérateurs de MEC sont calculés par des algorithmes itératifs, linéaires sur le nombre d'états et de transitions d'un automate. Bien que ces algorithmes soient applicables aux Arbres Partagés, ils ne sont pas assez efficaces à cause de la nature "ensembliste" (par opposition à "itérative") des Arbres Partagés. Il a donc été nécessaire de développer des algorithmes dédiés aux Arbres Partagés, travaillant de manière globale, dans le but de rendre les calculs plus rapides.

Dans cette section, nous passons en revue tous les algorithmes développés spécifiquement pour les Arbres Partagés. Ces algorithmes sont de deux types :

- Les algorithmes ensemblistes, qui utilisent les primitives ensemblistes de base définies sur les Arbres Partagés. Comme nous le verrons, ces algorithmes sont très fréquemment utilisés.
- Les algorithmes spécifiques, qui manipulent directement la structure interne des Arbres Partagés. Ces derniers sont peu utilisés, mais sont nécessaires pour rendre certaines opérations plus rapides.

4.3.1 Calcul des opérateurs élémentaires

Les quatre opérateurs élémentaires sont `src`(R), `tgt`(R), `rsrc`(Q), `rtgt`(Q) (voir Section 2.2.1).

Prenons l'exemple de `src`(R). Il s'agit de trouver l'ensemble des états d'un automate qui sont la source d'au moins une transition de l'ensemble R. La Figure 4.7 donne un exemple de calcul de `src` sur un Arbre Partagé (AP_R) contenant 4 multi-transitions. Le résultat est un Arbre Partagé (AP_{src}) contenant les 3 multi-états qui sont la source d'au moins une multi-transition.

Le principe de calcul de `src`(R) sur un Arbre Partagé multi-transitions est le suivant : il s'agit en fait d'un algorithme de copie d'un Arbre Partagé fonctionnant selon le principe de base (parcourt de haut en bas et construction de bas en haut), mais la valeur des noeuds de l'Arbre Partagé résultat est la valeur de l'état source

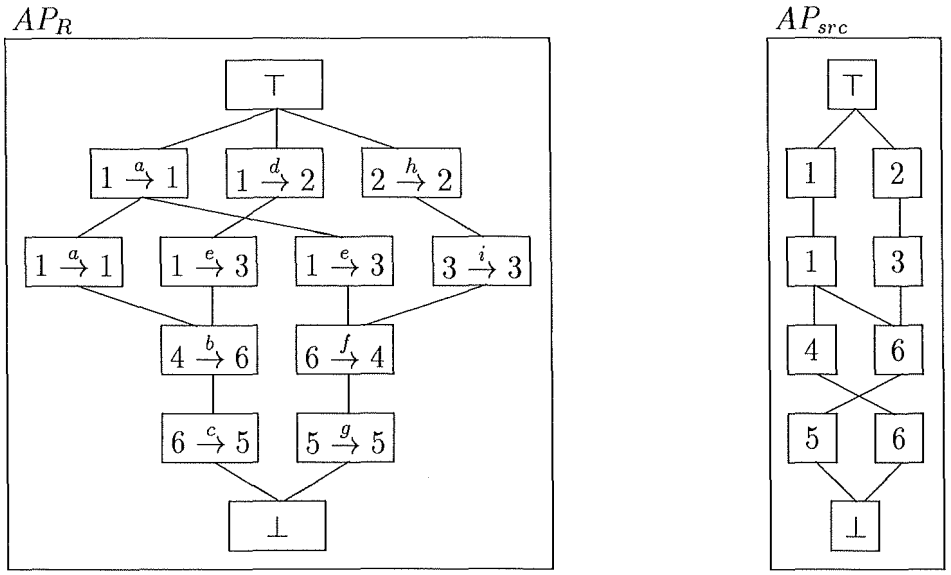


Figure 4.7: Exemple de calcul de l'opérateur `src` sur un Arbre Partagé

associé à la transition d'un noeud de l'Arbre Partagé en entrée. On effectue donc une restriction sur la $1^{ère}$ valeur de chaque triplet des noeuds transitions.

Il existe cependant une légère différence par rapport à un algorithme général: plusieurs noeuds "transitions" d'une même couche peuvent avoir le même état source (par exemple, les noeuds $1 \xrightarrow{a} 1$ et $1 \xrightarrow{a} 2$ de la couche 1 ont tous deux "1" comme état source). Dans ce cas, lorsque l'algorithme détecte qu'il ajoute une $2^{ème}$ fois le noeud $1_{AP_{src}}$ comme successeur de $\top_{AP_{src}}$, il effectue une union des deux sous-arbres avant insertion.

Le calcul de `tgt`(R) est identique au calcul de `src`(R): il suffit de considérer la $3^{ème}$ valeur du triplet (état but) et non plus la $1^{ère}$.

Les opérateurs `rsrc`(Q) et `rtgt`(Q) ont besoin de deux Arbres Partagés en entrée: un Arbre Partagé contenant les états de Q et un Arbre Partagé contenant l'ensemble des transitions de l'automate synchronisé. La primitive `rtgt`(Q) calcule l'ensemble des transitions d'un automate synchronisé dont l'états but appartient à l'ensemble Q.

Un exemple de calcul de `rtgt`(Q) est représenté en Figure 4.8. Le premier Arbre Partagé (AP_Q) contient un ensemble d'états, le deuxième (AP_T), un ensemble de transitions (nous considérons que c'est l'ensemble des transitions d'un automate synchronisé) et le troisième Arbre Partagé (AP_{rtgt}) contient le résultat du calcul de `rtgt`(Q).

Le principe de l'algorithme est toujours le même: le deux Arbres Partagés en entrée sont parcourus de haut en bas récursivement, chaque couple de noeuds

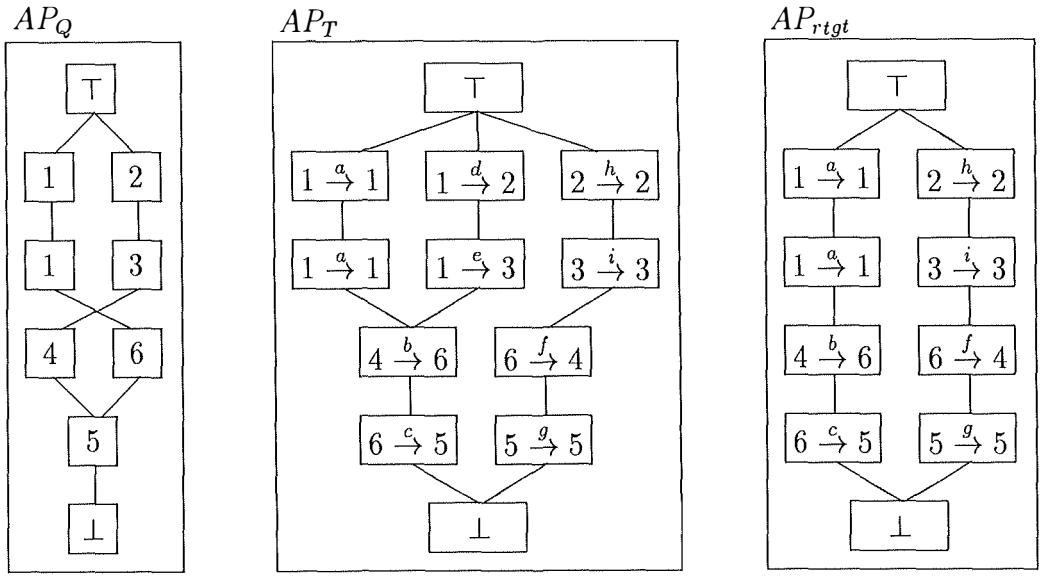


Figure 4.8: Exemple de calcul de l'opérateur `rtgt` sur deux Arbres Partagés

$\langle e_{tgt}, e_{src} \xrightarrow{t} e_{tgt} \rangle$ est pris en compte lors du parcours et les transitions $e_{src} \xrightarrow{t} e_{tgt}$ sont ajoutées de bas en haut dans l'Arbre Partagé résultat.

Pour le calcul de `rsrc`, on prend en compte chaque couple de noeuds $\langle e_{src}, e_{src} \xrightarrow{t} e_{tgt} \rangle$.

Remarque

Dans les algorithmes décrits dans les sections suivantes, nous aurons besoin de calculer l'équivalent de `rsrc`(Q) et `rtgt`(Q) avec un ensemble de transitions autre que l'ensemble des transitions de l'automate synchronisé. Nous noterons ces primitives respectivement `Rsrc`(Q, R) et `Rtgt`(Q, R), où Q est un ensemble quelconque d'états et R, un ensemble quelconque de transitions.

4.3.2 Calcul du produit synchronisé accessible

Le calcul du produit synchronisé accessible comprend trois étapes :

1. construction de l'Arbre Partagé des états initiaux;
2. construction de l'Arbre Partagé de la relation de transition globale;
3. calcul du produit accessible à partir des états initiaux en utilisant la relation de transition globale.

Construction de l'Arbre Partagé des états initiaux

L'ensemble des états initiaux de l'automate synchronisé est le produit cartésien des ensembles des états initiaux des automates composants. En général, chaque automate composant a un seul état initial, mais MEC offre la possibilité d'en avoir plusieurs, puisque les états initiaux sont définis à l'aide de la marque (obligatoire) `initial`.

Soit k automates $\langle \mathcal{A}_1, \ldots, \mathcal{A}_k \rangle$, d'états initiaux $\langle initial_1, \ldots, initial_k \rangle$, l'ensemble des états initiaux de l'automate synchronisé $\mathcal{A}$ sur les automates $\langle \mathcal{A}_1, \ldots, \mathcal{A}_k \rangle$ est :

$$initial = initial_1 \times \cdots \times initial_k$$

La construction de l'Arbre Partagé contenant les éléments de *initial* comprend deux étapes :

- On construit une couche par automate composant. Une couche i contient un noeud par élément de $initial_i$.
- On ajoute les noeuds $\top$ et $\bot$, puis on relie chaque noeud de chaque couche avec tous les noeuds de la couche inférieure[5].

Exemple

Soit les 4 automates :

- $\mathcal{A}_1$ d'états initiaux $initial_1 = \{1, 2\}$
- $\mathcal{A}_2$ d'états initiaux $initial_2 = \{1, 3\}$
- $\mathcal{A}_3$ d'états initiaux $initial_3 = \{4, 6\}$
- $\mathcal{A}_4$ d'état initial $initial_4 = \{5\}$

La Figure 4.9 représente les deux étapes de la construction de l'Arbre Partagé des états initiaux. L'ensemble des états de l'Arbre Partagé est bien :

$$\begin{aligned} \mathcal{E}lem(AP_{initial}) = \{ & (1,1,4,5),\ (1,1,6,5), \\ & (1,3,4,5),\ (1,3,6,5), \\ & (2,1,4,5),\ (2,1,6,5), \\ & (2,3,4,5),\ (2,3,6,5)\ \} \end{aligned}$$

[5] Un tel arbre partagé est dit *complet*, car le partage est maximum.

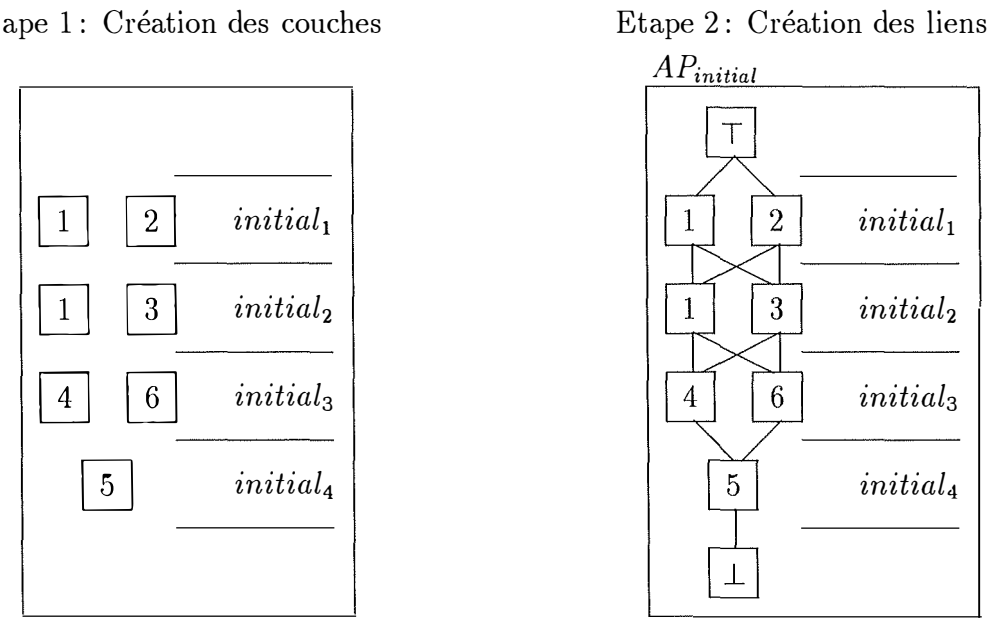


Figure 4.9: Construction d'un Arbre Partagé représentant des états initiaux

Construction de l'Arbre Partagé de la relation de transition globale

La relation de transition globale est l'ensemble des transitions du produit synchronisé tel qu'il a été défini en Section 2.1.2. Il s'agit donc de l'ensemble des multi-transitions définies par la contrainte de synchronisation.

Mathématiquement, on a la définition suivante : Soit un automate $\mathcal{A}$ à synchroniser sur les automates $\langle \mathcal{A}_1, \ldots, \mathcal{A}_k \rangle$ et l'ensemble des vecteurs de synchronisation $V = \{V_1, \ldots, V_n\}$ défini comme contrainte de synchronisation de $\mathcal{A}$, avec

$$V_i = (label_1^i, \ldots, label_k^i)$$

La relation de transition globale R est :

$$R = \bigcup_{i=1}^{n} R'(V_i)$$

avec

$$R'(V_i) = \{(e_1 \xrightarrow{label_1^i} e'_1, \ldots, e_k \xrightarrow{label_k^i} e'_k) : e_j \xrightarrow{label_j^i} e'_j \text{ pour } j = 1...k\}$$

L'algorithme de calcul de R se trouve en Figure 'Algorithme 4.1'. La construction de $R'(V_i)$ se fait en 2 étapes :

$R \leftarrow \{\}$
Pour $i \leftarrow 1$ jusque n
  Construire $R'(V_i)$
  $R \leftarrow R \cup R'(V_i)$
Fin

Algorithme 4.1: Calcul de la relation de transition globale

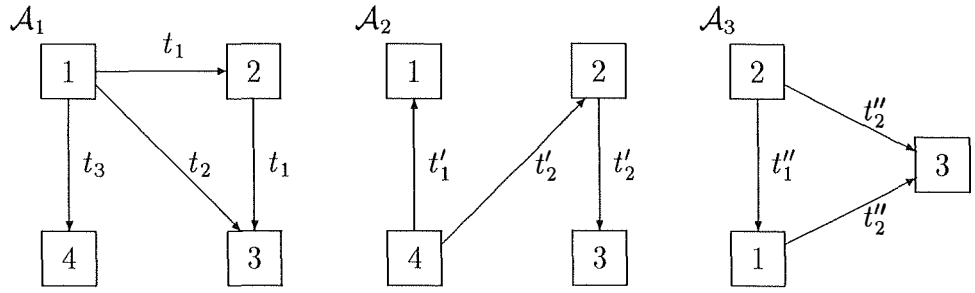


Figure 4.10: Exemple d'automates pour la construction de $R'(V_1)$

- On construit des noeuds dans les couches correspondant aux automates $\mathcal{A}_j$. Pour chaque transition de l'automate $\mathcal{A}_j$ dont le label est égal à $label_j^i$, on crée un noeud 'transition' dans la couche j.
- On ajoute les noeuds '$\top$' et '$\bot$', puis on relie tous les noeuds de chaque couche avec tous les noeuds de la couche suivante.

Exemple

Soit les trois automates de la Figure 4.10 et le vecteur de synchronisation :

$$V_1 = \{t_1, t'_2, t''_2\}$$

La Figure 4.11 représente les deux étapes de la construction de l'Arbre Partagé représentant $R'(V_1)$.

Calcul du produit accessible

Le calcul du produit accessible est effectué par un algorithme semi-naïf de recherche du point fixe d'un système d'équations. Historiquement, deux versions de l'algorithme ont été développées. Nous les présentons dans cette section.

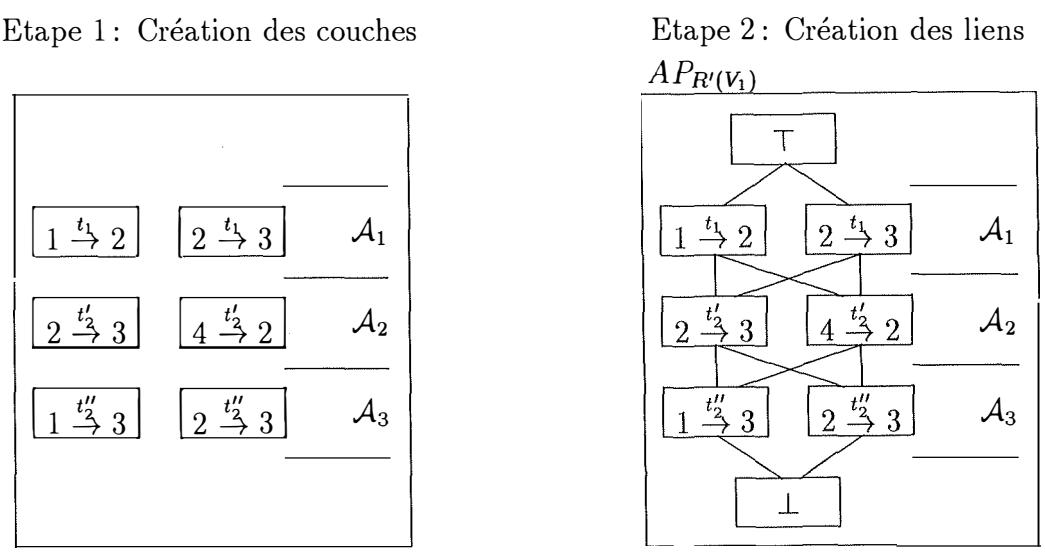


Figure 4.11: Construction de l'Arbre Partagé représentant $R'(V_1)$

```
TD ← initial
S ← initial
T ← {}
Tant que TD ≠ {} faire
    NS ← ReachableStates(TD, R)
    NT ← ReachableTransitions(TD, R)
    TD ← NS \ S
    S ← S ∪ NS
    T ← T ∪ NT
Fin
```

Algorithme 4.2: Algorithme semi-naïf de calcul du produit accessible (version 1)

Première version L'algorithme qui se trouve en Figure 'Algorithme 4.2' calcule le plus petit point-fixe du système d'équations suivant :

$$\begin{cases} S = initial \cup S \cup ReachableStates(R, S) \\ T = T \cup ReachableTransitions(R, S) \end{cases}$$

où *initial* est l'ensemble des états initiaux de l'automate à synchronisé et R, la relation de transition globale. L'algorithme est dit "semi-naïf" car il tient à jour un ensemble d'état TD (*To Develop*) qui doivent encore être développés. Le calcul est arrêté lorsque cet ensemble est vide.

Le but de l'algorithme est de calculer les états (S) et transitions (T) accessibles à partir de *initial* en appliquant les transitions de R.

La fonction $ReachableStates(TD, R)$ calcule l'ensemble des états accessibles en

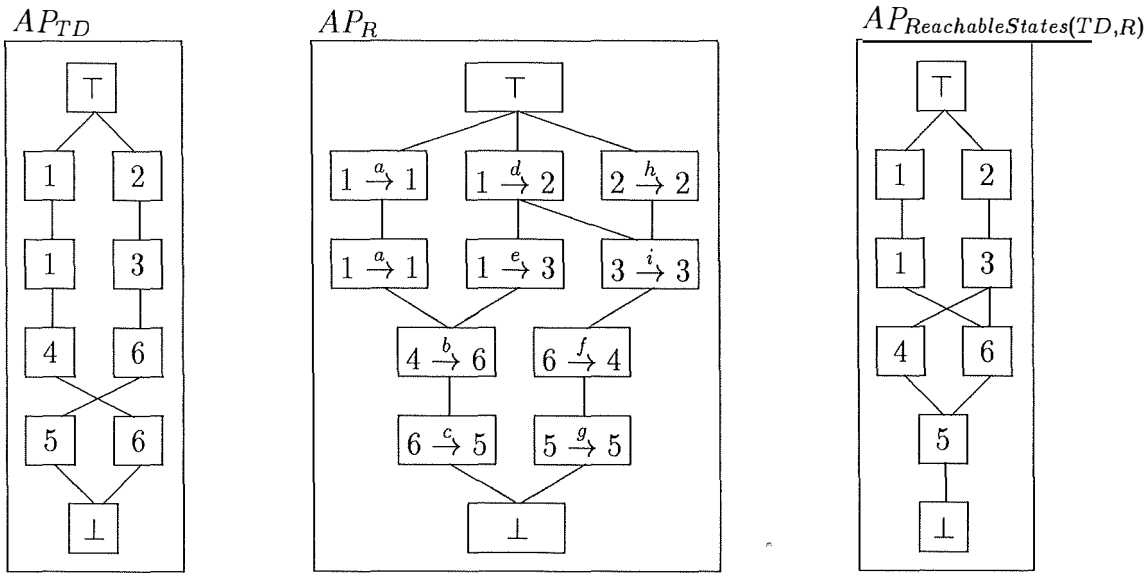


Figure 4.12: Exemple d'application de la fonction *ReachableStates*

une transition à partir de TD en utilisant les transitions de R. Le principe de la fonction est similaire à l'algorithme de calcul de `Rtgt`(TD, R), excepté que l'Arbre Partagé résultat est un Arbre Partagé contenant des états et non des transitions. Il s'agit en fait de calculer `tgt(Rsrc`(TD, R)). Un exemple de résultat de *ReachableStates*(TD, R) est donné en Figure 4.12.

La fonction *ReachableTransitions*(TD, R) calcule l'ensemble des transitions de R qui ont un état de TD comme source. Cet ensemble correspond aux transitions parcourues pour passer de TD à NS. Le principe de l'algorithme est similaire à celui du calcul de `Rtgt`(TD, R), avec TD et R comme Arbres Partagés en entrée. Un exemple de résultat de *ReachableTransitions*(TD, R) est donné en Figure 4.13.

Remarque

En pratique, pour accélérer les temps de calcul, les deux opérations sont effectuées en parallèle, c'est-à-dire que lorsque TD et R sont parcourus, les Arbres Partagés représentant *ReachableStates* et *ReachableTransitions* sont générés en parallèle. Pour réaliser cette opération, on a donc besoin de la memoization avec table de hachage double (voir Section 4.1.2).

Version améliorée Le précédent algorithme calcule à la fois les états et les transitions accessibles. Or, la définition du produit synchronisé accessible vue en Section 2.1.2, page 18, dit :

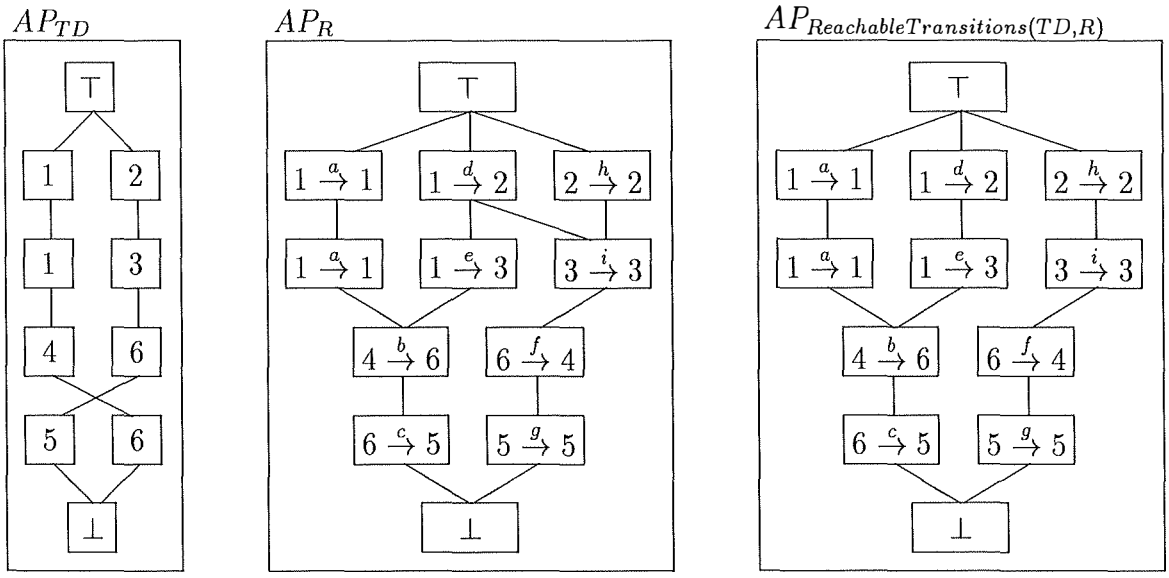


Figure 4.13: Exemple d'application de la fonction *ReachableTransitions*

MEC prend en compte seulement un sous-système de transitions du produit synchronisé : la partie accessible de celui-ci.

L'ensemble des états du système de transitions synchronisé est

$$initial \cup Reach(initial)$$

avec

- $initial = initial_1 \times \cdots \times initial_n$, où $initial_i$ est l'ensemble des états initiaux de $\mathcal{A}_i$,
- $Reach(initial)$ est l'ensemble des états accessibles à partir de l'ensemble $initial$.

L'ensemble des transitions du système de transitions synchronisé est le sous-ensemble de l'ensemble des transitions globales ayant leur source et leur cible dans l'ensemble des états du système.

La dernière partie de la définition indique qu'il n'est pas nécessaire de calculer les transitions accessibles itérativement. Il suffit de calculer T à partir de R (R est bien l'ensemble des transitions globales) une fois que S est définitif, par l'instruction suivante :

$$T \leftarrow \texttt{Rsrc}(S, R) \cap \texttt{Rtgt}(S, R)$$

$TD \leftarrow initial$
$S \leftarrow initial$
Tant que $TD \neq \{\}$ faire
  $NS \leftarrow ReachableStates(TD, R)$
  $TD \leftarrow NS \setminus S$
  $S \leftarrow S \cup NS$
Fin
$T \leftarrow \texttt{Rsrc}(S, R) \cap \texttt{Rtgt}(S, R)$

Algorithme 4.3: Algorithme semi-naïf de calcul du produit accessible (version 2)

$TD \leftarrow E_I$
$S \leftarrow \{\}$
Tant que $TD \neq \{\}$ faire
  $NS \leftarrow ReachableStates(TD, R_T)$
  $TD \leftarrow NS \setminus S$
  $S \leftarrow S \cup NS$
Fin

Algorithme 4.4: Calcul de $\texttt{reach}(E_I, R_T)$

$\texttt{Rsrc}(S, R)$ calcule l'ensemble des transitions de R qui ont au moins un état de S comme source. $\texttt{Rtgt}(S, R)$ calcule l'ensemble des transitions de R qui ont au moins un état de S comme cible. Il est donc évident que l'union de ces deux ensembles correspond à la définition de l'ensemble des transitions accessibles.

La nouvelle version de l'algorithme se trouve en Figure 'Algorithme 4.3'. Cette version calcule uniquement les états accessibles, puis l'ensemble T est calculé.

Le temps de calcul du produit accessible est donc divisé par facteur d'environ 2, puisque seul l'ensemble S est calculé itérativement. Comme nous le verrons au Chapitre 6, ce facteur est même un peu supérieur, car, en général, l'Arbre Partagé des transitions est plus gros, et donc plus coûteux à calculer.

4.3.3 Calcul des états accessibles

La primitive $\texttt{reach}(E_I, R_T)$ de calcule l'ensemble des états accessibles à partir des états de l'ensemble E_I en appliquant les transitions de R_T.

L'algorithme de calcul de `reach` dédié aux Arbres Partagés se trouve en Figure 'Algorithme 4.4'. Il est du même type que celui du calcul du produit accessible, excepté qu'on utilise R_T comme relation de transition (et non pas la relation de transition globale) et qu'il ne faut calculer que les états accessibles.

$TD \leftarrow E_I$
$S \leftarrow \{\}$
Tant que $TD \neq \{\}$ faire
  $NS \leftarrow CoreachableStates(TD, R_T)$
  $TD \leftarrow NS \setminus S$
  $S \leftarrow S \cup NS$
Fin

Algorithme 4.5: Calcul de `coreach`(E_I, R_T)

4.3.4 Calcul des états co-accessibles

L'algorithme de calcul de `coreach`(E_I, R_T) se trouve en Figure 'Algorithme 4.5'. Il est identique à l'algorithme de calcul de `reach`, excepté qu'on utilise la fonction $CoreachableStates$. Il s'agit de trouver les états co-accessibles à partir de TD en une transition de R_T.

4.3.5 Calcul des plus courts chemins

L'opérateur `trace`(E_I, R_T, E_F) de MEC permet de calculer un plus court chemin entre les états initiaux de E_I et les états finaux de E_F, en appliquant les transitions de R_T.

L'algorithme de MEC vu au Chapitre 3 ne permet pas d'obtenir des performances satisfaisantes avec les Arbres Partagés, car :

1. il est linéaire sur le nombre d'états et de transitions du système ;
2. il requiert l'utilisation d'un compteur d'états, ce qui est très inefficace sur les Arbres Partagés.

L'algorithme dédié aux Arbres Partagés se trouve en Figure 'Algorithme 4.6'. Il a été proposé par *B. Le Charlier*[5]. Il est quasiment identique à la première version de l'algorithme de calcul du produit accessible vu en Section 4.3.2.

L'idée est de pouvoir identifier à quelle itération un ensemble de transitions a été parcouru, pour pouvoir reconstituer un des chemins parcourus lorsqu'un état de E_F est atteint.

Pour ce faire, il suffit d'ajouter un élément à chaque n-uplet "multi-transitions" contenant le numéro de l'itération où les multi-transitions sont générées par la fonction $ReachableTransitions$. C'est le rôle de la fonction $AjouterLongueur(NT, Itération)$, qui ajoute une couche dans NT en-dessous du noeud '$\top$'. Cette couche contient un noeud de valeur "*Itération*". Tous les n-uplets de NT sont donc identifiés par le numéro d'itération. Ainsi, lors de l'union

$TD \leftarrow E_I$
$S \leftarrow E_I$
$T \leftarrow \{\}$
$Itération \leftarrow 1$
Tant que $TD \neq \{\}$ faire
  $NS \leftarrow ReachableStates(TD, R_T)$
  $NT \leftarrow ReachableTransitions(TD, R_T)$
  $TD \leftarrow NS \setminus S$
  $S \leftarrow S \cup NS$
  $AjouterLongueur(NT, Itération)$
  $T \leftarrow T \cup NT$
  Si $S \cap E_F \neq \{\}$ alors
    Stop `/* Chemin trouvé */`
  $Itération \leftarrow Itération + 1$
Fin

Algorithme 4.6: Algorithme de calcul de `trace`(E_I, R_T, E_F)

de T et de NT, on ajoute à T un sous-arbre dont le noeud racine à la valeur de l'itération en cours.

Si l'algorithme trouve un chemin ($S \cap E_F \neq \{\}$), il suffit de prendre une transition du sous-arbre de longueur maximale qui a un état de E_F comme cible, puis de parcourir tous les sous-arbres cl' "*Itération*" décroissante en choisissant à chaque fois une transition qui a comme cible la source de la transition trouvée lors du choix précédent.

4.3.6 Calcul des composantes fortement connexes

La primitive `loop`(R, R') recherche les composantes fortement connexes de R' contenant au moins une transition de R.

Rappelons la définition formelle de `loop`: Une transition $t \in$ `loop`(R, R') si et seulement si t appartient à un chemin p non vide tel que:

1. l'origine de p est égale à sa destination,
2. chaque transition de p appartient à R',
3. au moins une transition de p appartient à R.

L'algorithme linéaire de MEC ne permet pas d'obtenir des performances satisfaisantes avec les Arbres Partagés, car:

1. il est linéaire sur le nombre de transitions du système;

2. il requiert l'utilisation d'un compteur d'états, ce qui est très inefficace sur les Arbres Partagés.

L'algorithme dédié aux Arbres Partagés, proposé par *B. Le Charlier*, considère trois types de sous-graphes dans un graphe :

Les grappes

Une grappe G est un ensemble de transitions tel que :

$$\forall t_1, t_2 \in G, t_1 \neq t_2 : \exists \text{ un chemin } p \subset G : \\ \alpha(p) = \alpha(t_1) \wedge \beta(p) = \beta(t_2)$$

Les chemins entre grappes

Un chemin entre grappes est un chemin qui mène d'une grappe à une autre, sans "revenir en arrière". Un chemin p est un chemin entre grappes ssi p ne contient pas de grappe et

$$\exists G_1 \neq G_2 \subset R' : \alpha(p) \subset Etats(G_1) \wedge \beta(p) \subset Etats(G_2)$$

avec $Etats(G) = \mathtt{src}(G) \cap \mathtt{tgt}(G)$

Les filaments

Un filament f est un chemin dont l'origine n'a pas de prédécesseur ou dont la destination n'a pas de successeur.

L'idée de l'algorithme (voir Figure 'Algorithme 4.8') est de boucler tant que R' n'est pas vide en enlevant dans un premier temps les filaments (voir Figure 4.7), puis en choisissant une transition t dans R' restant. Ensuite, on calcule les transitions accessibles à partir de la destination de t et les transitions co-accessibles à partir de la source t. Si t se trouve dans une grappe, l'intersection des deux ensembles constitue la grappe (qui est enlevée de R' et ajoutée dans Rr'), sinon on t fait partie d'un chemin et on le 'brise' en enlevant t de R'. On continue à itérer jusqu'à ce que R' soit vide. Le résultat du calcul est l'ensemble des transitions de Rr'. On prend en compte R lors du choix de t, qui doit se faire dans $R' \cap R$.

Il est a noter que l'algorithme permet aisément de compter le nombre de grappes (à chaque fois que $GR \neq \{\}$, on a trouvé une nouvelle grappe).

Un premier point critique de cet algorithme est qu'il est fortement dépendant du nombre de grappes dans R'. Plus ce nombre est élevé, plus l'algorithme dégénère.

Un second point critique est le choix de la transition t. Si ce choix est fait arbitrairement (par exemple, toujours la première transition) dans une 'zone' de l'Arbre Partagé où toutes les transitions constituent un chemin de longueur 1, l'algorithme devient linéaire sur le nombre de transitions. Une stratégie non-déterministe (basée sur l'expérimentation) a donc été mise en oeuvre. Elle est la suivante :

```
Rr' ← R'
Repeter
   E1 ← src(Rr') \ tgt(Rr')
   E2 ← tgt(Rr') \ src(Rr')
   /* E1 = états cibles d'aucune transition de Rr' */
   /* E2 = états sources d'aucune transition de Rr' */
   Si E1 = {} et E2 = {}
      Sortir de la boucle
   Rr' ← Rr' \ (Rsrc(E1, Rr') ∪ Rtgt(E2, Rr'))
   /* Les transitions éliminées ne peuvent pas appartenir
      à loop car soit leur source n'est pas atteignable
      soit leur destination ne mêne à aucun état */
Fin
R' ← Rr'
```

Algorithme 4.7: Algorithme de calcul de $EnleverFilaments(R')$

```
Rr' ← {}
Tant que R' ≠ {}
   EnleverFilaments(R')
   Si (R' ∩ R) = {}
      Sortir de la boucle
   Choisir t dans (R' ∩ R)
   Calculer A : transitions accessibles à partir
                de la destination de t
   Calculer B : transitions à partir desquelles on
                peut accéder à la source de t
   GR ← A ∩ B
   Si GR ≠ {} alors
      R' ← R' \ GR
      Rr' ← Rr' ∪ GR
   Sinon
      R' ← R' \ t
   Fin
Fin
```

Algorithme 4.8: Algorithme de calcul de $\texttt{loop}(R, R')$

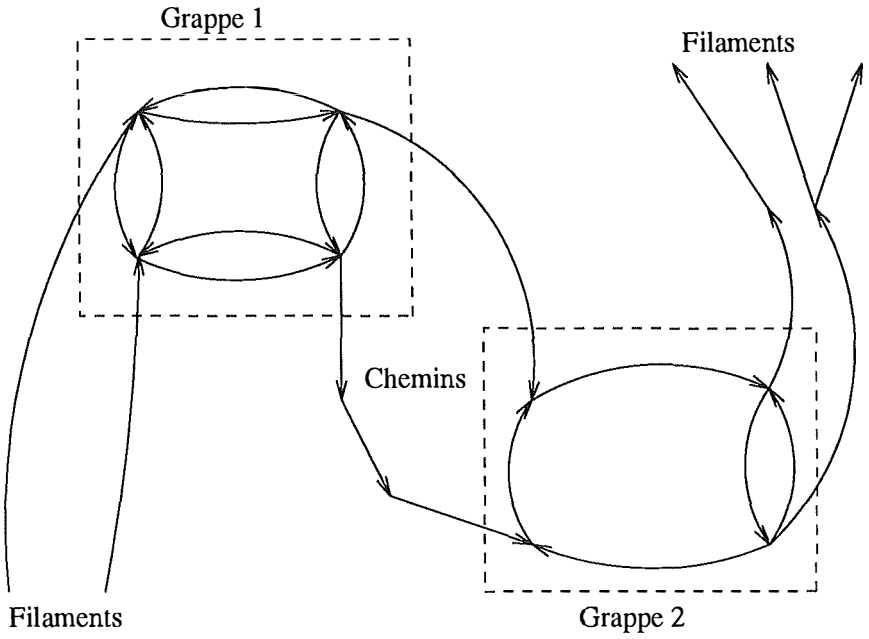


Figure 4.14: Exemple de graphe pour le calcul de `loop`

1. On commence par choisir t au milieu de l'Arbre Partagé ;
2. Si t était sur un chemin de longueur 1 lors du choix précédent, on choisit t dans la partie gauche de l'Arbre Partagé ;
3. Si t était sur un chemin de longueur 1 lors du choix précédent, on choisit t dans la partie droite de l'Arbre Partagé ;
4. Si t était sur un chemin de longueur 1 lors du choix précédent, on choisit t à au hasard dans l'Arbre Partagé ;
5. Si t était sur un chemin de longueur 1 lors du choix précédent, on recommence en 1.

Exemple

Dans la Figure 4.14, on trouve un exemple de graphe ayant 2 grappes, 2 chemins et 5 filaments. La première étape de l'algorithme consiste a retirer les filaments du graphe (Figure 4.15).

Ensuite, il faut choisir une transition t dans le graphe restant (en fait dans le graphe restant $\cap R$, car il faut tenir compte de R). On calcule A, l'ensemble des transitions accessibles à partir de $\beta(t)$, puis on calcule B, l'ensemble des transitions à partir desquelles on peut arriver à $\alpha(t)$.

On calcule ensuite $A \cap B$. Deux cas sont alors possibles :

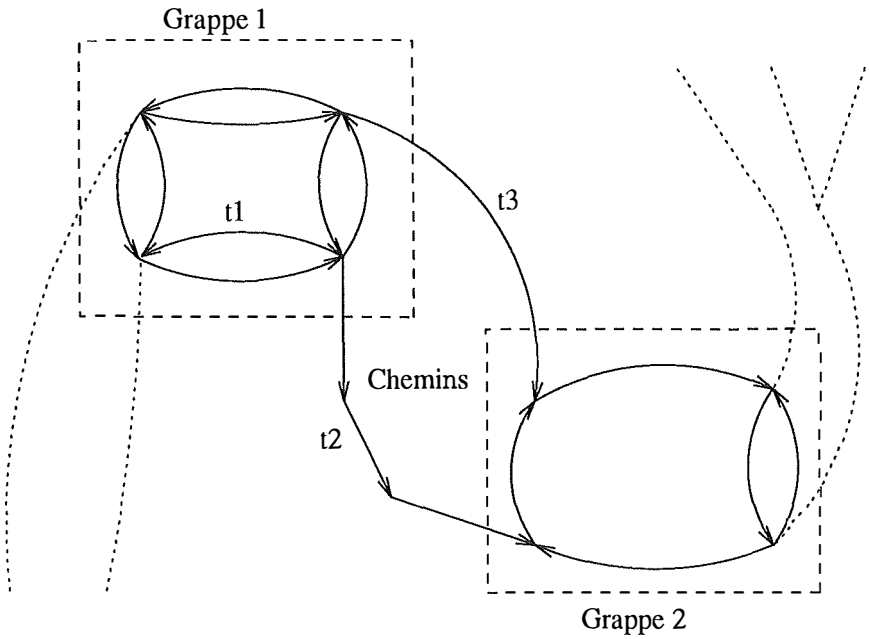


Figure 4.15: Exemple de graphe où on a retiré les filaments

1. On a choisi t dans une grappe (soit **t1** dans **Grappe 1** dans la Figure 4.15). L'intersection $A \cap B$ contient alors les transitions de **Grappe 1**. On a donc déterminé une nouvelle grappe, que l'on peut ajouter dans le résultat du calcul (les transitions trouvées répondent bien à la définition de `loop`). En retirant **Grappe 1** de l'ensemble des transitions, on crée de nouveaux filaments (Figure 4.16).
2. On a choisi t sur un chemin (soit **t2** dans la Figure 4.15). Dans ce cas, on retire t de l'ensemble des transitions. On crée ainsi deux nouveaux filaments (Figure 4.17).

Une fois l'un des deux cas traité, on peut recommencer à itérer pour supprimer les nouveaux filaments.

Il est à noter qu'il existe des cas où on ne créera pas de nouveaux filaments. Si, par exemple, on avait choisi t égal à **t3**, qui se trouve sur un chemin de longueur 1, aucun nouveau filament n'est trouvé (Figure 4.18). Dans ce cas (qui est impossible à déterminer à priori), la recherche de nouveaux filaments à l'itération suivante du calcul ne permettra pas de diminuer la taille du graphe.

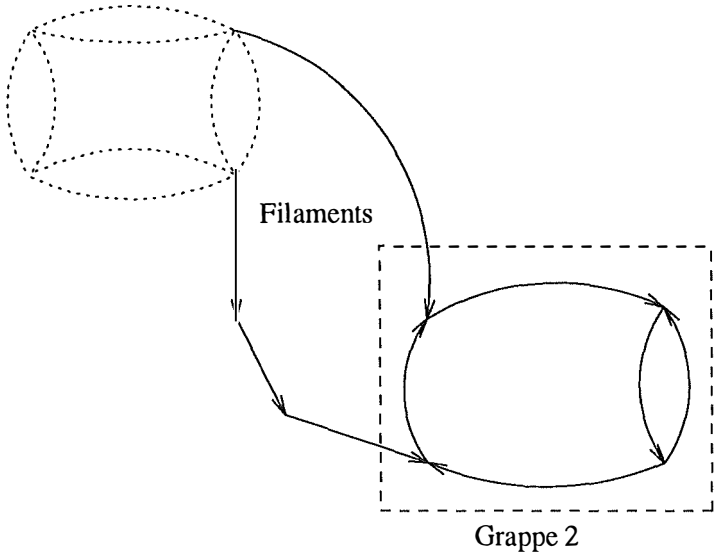


Figure 4.16: Exemple de graphe dont on a retiré une grappe

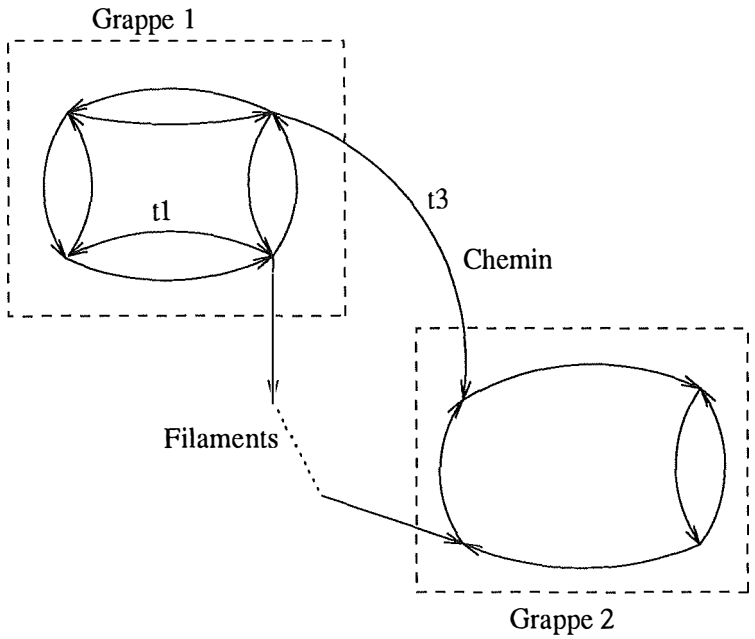


Figure 4.17: Exemple de graphe dont on a retiré un chemin

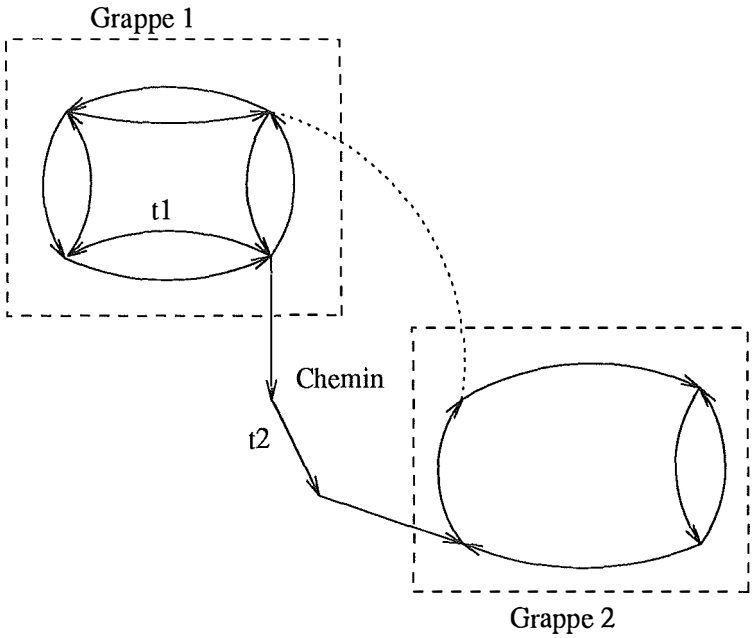


Figure 4.18: Exemple de graphe dont on a retiré un chemin de longueur 1

4.3.7 Calcul des points fixes d'un système d'équations

L'algorithme linéaire de MEC ne permet pas d'obtenir des performances satisfaisantes avec les Arbres Partagés, car :

1. il est linéaire sur le nombre de d'états et de transitions du système ;
2. il requiert l'utilisation d'un compteur d'états, ce qui est très inefficace sur les Arbres Partagés.

L'algorithme dédié aux Arbres Partagés est une version simplifiée de l'algorithme général de calcul du point fixe d'une fonction monotone proposé en [6].

Cet algorithme est général, car il ne requiert qu'un seul lien avec le problème à traiter (dans ce cas, la résolution d'un système d'équations): une fonction d'évaluation d'une équation. De plus, cet algorithme permet d'éviter de nombreux calculs superflus grâce à l'utilisation d'un graphe de dépendance qui associe à chaque variable l'ensemble des variables à re-calculer en cas de modification.

Structure de données

A chaque variable du système d'équations à résoudre correspond une marque d'état ou de transitions suivant le type de la variable. Comme nous travaillons avec des Arbres Partagés, ces marques sont en fait des Arbres Partagés.

En plus des marques, l'algorithme a besoin des structures de données suivantes :

***Inititiated Computation Set* (*ics*)**
ics est un tableau de 'booléens' dont la taille est égale au nombre de variables du système. Une entrée i de *ics* est à $VRAI$ si la variable i correspondante est en cours de calcul.

***Dependency Graph* (*dg*)**
dg est un tableau de 'listes d'entiers' dont la taille est égale au nombre de variables du système. Une entrée i de *dg* contient la liste des variables à recalculer en cas de modification de la variable i.

***Domain of Dependency Graph* (*dom_dg*)**
dom_dg est un tableau de 'booléens' dont la taille est égale au nombre de variables du système. Une entrée i de *dom_dg* est à $VRAI$ si la variable correspondante est dans le domaine, c'est-à-dire que les éléments de *dg[i]* sont valides et représentent la liste des variables à recalculer en cas de modifications de la variable i.

La fonction Insert_dg

Insert_dg(x, y) insère y dans le graphe de dépendance de x.

La fonction Remove_dg

Remove_dg(x) retire x et tous ses dépendants (récursivement) du graphe de dépendance

La fonction Eval_var

Eval_var(x, exp) évalue une variable (appel de *Compute*(x)) puis l'insère dans le graphe de dépendance de exp (appel de *Insert_dg*(exp, x))

La fonction Eval_exp

Eval_exp(x) appelle *Eval_var*(y, x) pour toutes les variables y présentes dans le terme de l'équation de x. Ensuite, elle renvoie le résultat de l'évaluation du terme sur la nouvelle valeur des variables.

La fonction Compute

Compute(x) calcule la meilleure approximation possible de la variable x. L'algo-

```
Si dom_dg[x] alors
   /* x est optimal pour le moment */
Sinon
   Si ics[x] alors
      /* x est déjà en cours de calcul */
   Sinon
      ics[x] ← VRAI
      Répeter
         dom_dg[x] ← VRAI
         x_new ← Eval_Exp(x)
         Si #(x_new) = #(x) alors
            /* x_new n'a pas été amélioré */
         Sinon
            x ← x_new
            Remove_dg[x]
         Fin
      Tant que dom_dg[x] = FAUX
      ics[x] ← FAUX
   Fin
Fin
```

Algorithme 4.9: Calcul de $Compute(x)$

rithme de $Compute(x)$ se trouve en Figure 'Algorithme 4.9'.

L'algorithme du calcul du point fixe

L'algorithme du calcul du point fixe se trouve en Figure 'Algorithme 4.10'. n est le nombre d'équations du système. L'initialisation d'une variable selon son signe a été expliqué en Section 2.3.2.

```
Pour i ← 1 jusque n
   Initialiser xi selon son signe
   dom_dg[i] ← FAUX
   ics[i] ← FAUX

Pour i ← 1 jusque n
   Compute(x_i)
```

Algorithme 4.10: Calcul du point fixe d'un système d'équations

4.3.8 Evaluation des expressions de comparaison de labels

MEC permet l'évaluation de toute une série d'expressions sur les noms des états et les labels des transitions. Il a été nécessaire de créer des algorithmes dédiés aux Arbres Partagés, toujours dans le but d'améliorer l'efficacité des algorithmes génériques de MEC. Les liste des expressions de 'comparaisons de labels' a été donnée en Section 2.2.5.

Dans les sous-sections qui suivent, seules les expressions de calcul d'égalité de type 'états' sont expliquées. Il est cependant aisé de généraliser aux calculs sur les transitions et de différence (ce qui a été fait lors de l'implémentation).

En ce qui concerne l'implémentation des algorithmes d'évaluation de ces expressions, tous sont exécutés via une fonction de restriction (*matching*) paramètrable. Cette fonction accepte un Arbre Partagé de transitions ou d'états en entrée, utilise ou pas la memoization, peut appeler une fonction spécifique lors de l'entrée et lors de la sortie de chaque noeud parcouru, ainsi qu'une fonction booléenne appelée lorsqu'une opération de restriction sur la valeur d'un noeud est nécessaire. Tous ces paramètres sont instantiés de manière différente pour chaque type d'expression à évaluer.

Expressions `!state = PATTERN`

L'algorithme général le plus efficace trouvé pour calculer ce genre d'expression est linéaire sur le nombre d'états de l'Arbre Partagé. Ceci est dû à la présence possible de 'jokers' dans '`PATTERN`', qui ne permettent par, en général, d'utiliser la memoization lors du parcours de l'arbre.

L'algorithme effectue un parcours lexicographique de tous les éléments de l'arbre, en maintenant à jour une chaîne de caractère avec le nom des états composants déjà parcourus. Lorsque le dernier niveau de l'arbre est atteint, la chaîne construite est comparée avec '`PATTERN`' ; la fonction de comparaison tient compte des 'jokers' éventuels dans '`PATTERN`').

Lorsque l'algorithme remonte d'un niveau (quand tous les fils du niveau inférieur ont été visités), la chaîne construite est réinitialisée à sa valeur lors de l'arrivée dans le niveau courant.

Au niveau implémentation, le calcul est effectué avec la fonction de restriction, sans memoization. La fonction d'entrée dans un noeud mémorise la longueur courante de la chaîne. La fonction de sortie réinitialise la chaîne à cette longueur. L'opérateur de restriction n'effectue une comparaison de chaînes que lorsqu'il est appelé sur un noeud du dernier niveau de l'arbre.

Expressions `!state[n] = PATTERN`

L'algorithme utilisé pour évaluer ce type d'expression est un cas particulier du précédent. On peut ici utiliser la memoization, puisque la comparaison est limitée à une couche de l'arbre.

Au niveau implémentation, le calcul est effectué avec la fonction de restriction, avec memoization. Il n'y a pas de fonction d'entrée dans un noeud, ni de fonction de sortie. L'opérateur de restriction effectue une restriction uniquement pour les noeuds de la couche n de l'arbre. Il compare le nom de l'état correspondant au noeud courant avec '`PATTERN`'.

Expressions `!state[n] =!state[m]`

L'algorithme utilisé pour évaluer ce type d'expression est un cas plus général de l'algorithme de la section précédente. L'idée est d'appeler l'algorithme de la section précédente pour chaque valeur de `!state[n]` qui est égale à une valeur de `!state[m]`. Il suffit ensuite de faire une union de tous les Arbres Partagés ainsi obtenus.

4.3.9 Test d'appartenance à des sous-marques

Dans MEC, il est possible de demander l'évaluation d'une expression du type `sous_marque[`i`]` (voir Section 2.2.6). Cette expression désigne un sous-ensemble des états ou transitions du l'automate synchronisé courant dont la $i^{ème}$ composante appartient à la marque `sous_marque`, définie sur le $i^{ème}$ automate composant.

Au niveau implémentation, le calcul est effectué avec la fonction de restriction, avec memoization. Il n'y a pas de fonction d'entrée dans un noeud, ni de fonction de sortie. L'opérateur de restriction effectue une restriction uniquement pour les noeuds de la couche i de l'arbre. Il vérifie si l'état (ou la transition) correspondant au noeud courant appartient aux états (ou aux transitions) de la marque '`sous_marque`' du $i^{ème}$ automate composant.

4.3.10 Calcul des projections

Une primitive de MEC, que nous n'avons pas encore signalée, permet d'effectuer des projections de marques d'un automate synchronisé. Le résultat d'une projection est une marque créée dans l'automate composant correspondant à l'indice de la projection.

La projection d'un Arbre Partagé sur sa $i^{ème}$ composante est l'ensemble des

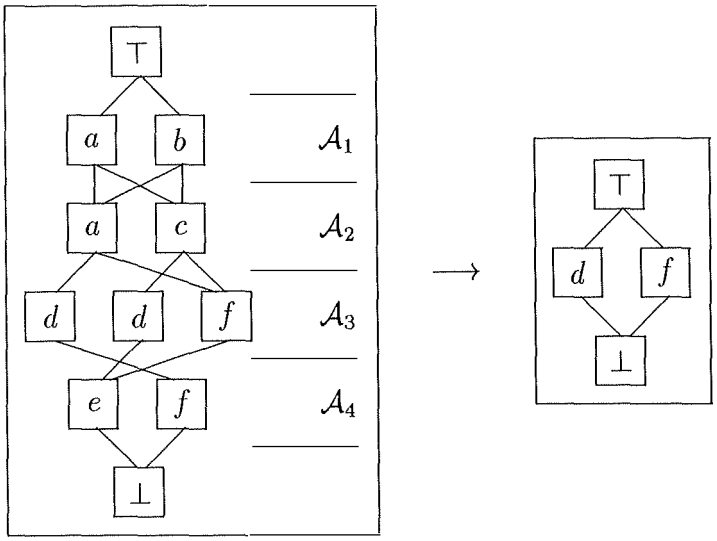


Figure 4.19: Exemple de projection d'un Arbre Partagé

noeuds de la $i^{ème}$ couche. D'un point de vue technique, il faut bien entendu éviter les noeuds 'doublons' dans la couche et assurer la création de la marque dans l'automate composant.

Exemple

La Figure 4.19 montre comment effectuer la projection d'un Arbre Partagé sur sa $3^{ème}$ composante.

4.4 Améliorations de la gestion des Arbres Partagés

Comme nous l'avons déjà signalé, les algorithmes de base de gestion de Arbres Partagés ont été fournis sous forme de bibliothèque. Un premier travail a été de transformer tous les algorithmes du langage "Pascal" en langage "C", pour une meilleure intégration dans MEC.

Cependant, après certaines expérimentations (voir Chapitre 6), il s'est avéré que ces algorithmes pouvaient encore être améliorés. En fait, les algorithmes avaient tendance à dégénérer (en terme de temps de calcul) quadratiquement avec la taille des Arbres Partagés.

Dans cette section, nous expliquons deux améliorations qui ont été implémentées dans la bibliothèque. Ces améliorations ne sont pas de même nature: la première

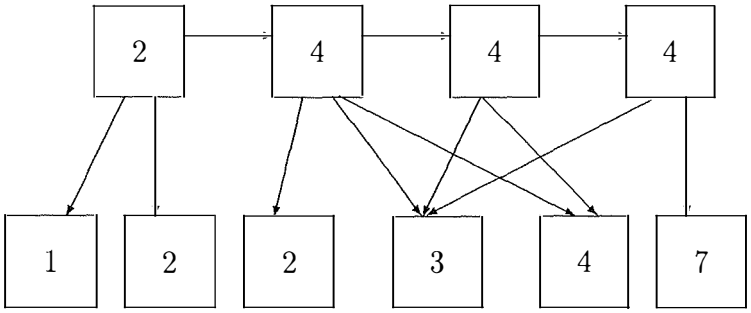


Figure 4.20: Exemple de couche d'un Arbre Partagé

concerne la gestion interne de la structure des Arbres Partagés (la gestion des noeuds dans une couche), tandis que la deuxième consiste en la création d'une nouvelle primitive qui accélère certains algorithmes (l' *union incrémentale*).

4.4.1 Gestion des noeuds dans une couche

Comme nous l'avons expliqué dans la Section 4.1.2, page 43, tous les algorithmes de construction d'un Arbre Partagé fonctionne selon un principe "bottom-up" : l'arbre créé est construit de bas en haut, de telle sorte qu'au moment d'insérer un noeud dans une couche, l'ensemble de ses successeurs est connu. Il est ainsi possible de vérifier s'il n'existe pas déjà un noeud ayant la même valeur et le même ensemble de successeurs. Si c'est le cas, le noeud à insérer est détruit et le noeud existant est pris en compte. Ce principe permet de garantir la canonicité de l'Arbre Partagé construit (règle 2 de la définition d'un Arbre Partagé).

Dans la première version de la bibliothèque des Arbres Partagés, les noeuds d'une couche sont reliés par une liste chaînée, c'est-à-dire que chaque noeud contient un pointeur vers le noeud suivant dans la couche. La liste est triée sur la valeur des noeuds, mais il arrive très souvent, dans des Arbres Partagés de taille moyenne ou importante, que beaucoup de noeuds de même valeur avec des successeurs différents se trouvent dans une couche. La Figure 4.20 représente un exemple de couche d'un Arbre Partagé contenant trois noeuds de valeur '4'.

A chaque insertion d'un nouveau noeud dans une couche, il faut donc parcourir la liste des noeuds de la couche[6] pour vérifier s'il existe un noeud équivalent, c'est-à-dire un noeud ayant la même valeur et le même ensemble de successeurs. Le coût en terme de temps de calcul de tous ces parcours n'est pas négligeable pour deux raisons :

[6]Pour être précis, il ne faut parcourir, en moyenne, que la moitié de la liste, puisqu'elle est triée.

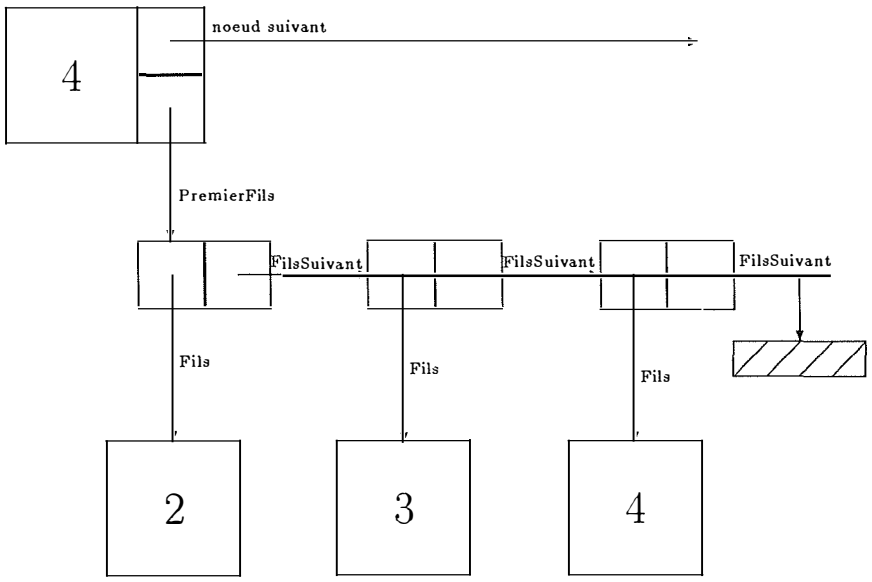


Figure 4.21: Exemple de structure d'un noeud d'un Arbre Partagé

1. La fonction d'insertion d'un noeud dans une couche peut être exécutée un très grand nombre de fois. Par exemple, pour calculer certains produits synchronisés accessibles complexes, plusieurs millions d'appels à la fonction sont effectués, ce qui peut représenter jusqu'à 40% du temps de calcul total.

2. Le test d'équivalence de deux noeuds de même valeur est une opération assez coûteuse, puisqu'il faut comparer les deux ensembles de successeurs[7].

   En pratique, un ensemble de successeurs d'un noeud est représenté par une liste chaînée de pointeurs vers les noeuds successeurs. Cette liste est triée sur la valeur des noeuds successeurs. La Figure 4.21 représente le noeud de valeur '4', ayant des noeuds successeurs de valeur '2', '3'et '4'. Chaque élément de la liste contient un pointeur vers un noeud successeur (pointeur `Fils`) et un pointeur vers l'élément suivant de la liste (pointeur `FilsSuivant`). Le premier élément de la liste (pointeur `PremierFils`) se trouve dans un champ spécial de la structure 'noeud'.

   Cette liste intermédiaire entre un noeud d'une couche et ses successeurs dans la couche suivante est nécessaire, car un noeud peut être présent dans l'ensemble des successeurs de plusieurs noeuds différents. Par exemple, dans la Figure 4.20, le noeud de valeur '3' fait partie des successeurs de trois noeuds différents.

   Pour vérifier que deux noeuds ont un même ensemble de successeurs, il

[7]Si les noeuds ont des valeurs différentes, on peut immédiatement dire que les noeuds ne sont pas équivalents.

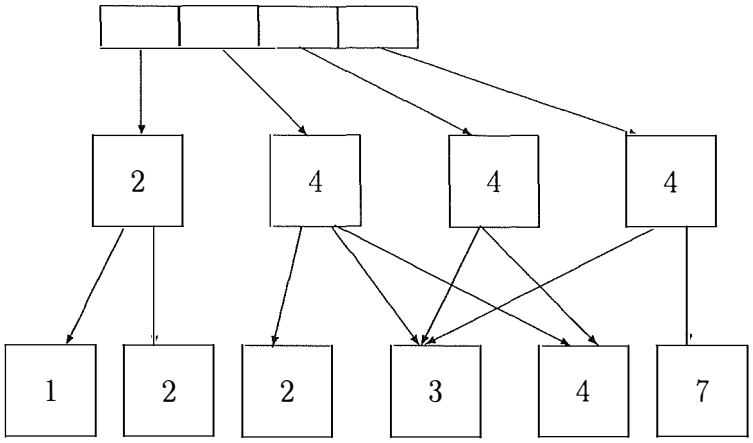


Figure 4.22: Exemple de couche d'un Arbre Partagé avec tableau de pointeurs

faut donc parcourir les deux listes de successeurs en parallèle et vérifier que les valeurs des pointeurs '`Fils`' sont identiques. Dès que deux `Fils` sont différents, ou que la fin d'une des deux listes est atteinte, sans que la fin de l'autre ne le soit, on peut déduire que les deux noeuds ne sont pas équivalents. Cette déduction est valide étant donné qu'un pointeur `Fils` identifie de manière unique un noeud successeur et que les listes des noeuds successeurs sont triées sur la valeur de ces noeuds.

Pour améliorer la gestion des noeuds dans une couche, et donc diminuer le temps de calcul imputable à la fonction d'insertion d'un noeud, la liste chaînée a été remplacée par un tableau de pointeurs. Ainsi, au lieu de lier chaque noeud d'une couche avec le noeud suivant dans la couche, la couche contient un tableau de pointeurs vers ses noeuds (voir Figure 4.22). L'idée est de trier ce tableau et ainsi remplacer l'algorithme de recherche linéaire dans la liste par un algorithme de recherche dichotomique dans le tableau.

Le principal problème à résoudre pour implémenter cette amélioration est la création d'une fonction 'C' de comparaison de deux noeuds, qui puisse non seulement tester l'équivalence des noeuds, mais également tester si un noeud est "plus petit" ou "plus grand" qu'un autre. Cette fonction est nécessaire pour l'algorithme de recherche dichotomique.

La spécification de la fonction *ComparerNoeuds*$(n1, n2)$ à réaliser est la suivante :

Entrée $n1$ et $n2$ sont les deux noeuds à comparer
Sortie Le résultat de la fonction est :

$$\begin{cases} = 0 \text{ si } n1 = n2 \\ < 0 \text{ si } n1 < n2 \\ > 0 \text{ si } n1 > n2 \end{cases}$$

avec

- $n1 = n2$ ssi $val(n1) = val(n2) \wedge succ(n1) = succ(n2)$
- $n1 < n2$ ssi

$$\begin{array}{l} val(n1) < val(n2) \\ \vee\ \#succ(n1) < \#succ(n2) \\ \vee\ (\exists i \in \{1, \ldots, \#succ(n1)\} : fils(n1, i) < fils(n2, i)) \end{array}$$

- $n1 > n2$ ssi

$$\begin{array}{l} val(n1) > val(n2) \\ \vee\ \#succ(n1) > \#succ(n2) \\ \vee\ (\exists i \in \{1, \ldots, \#succ(n1)\} : fils(n1, i) > fils(n2, i)) \end{array}$$

et $fils(n, i)$ est la valeur du $i^{ème}$ pointeur `Fils` de la liste des successeurs du noeud n.

L'implémentation "C" de la fonction se trouve en Figure 'Algorithme 4.11'. Le principe de fonctionnement est le suivant:

1. Si `val(n1) != val(n2)`, le résultat est `rc = val(n1) - val(n2)`.
2. Sinon, il faut considérer les successeurs. On parcourt la liste des successeurs de `n1` et `n2` en parallèle et on s'arrête dès qu'on atteint la fin d'une des deux listes (`s1 == NULL` ou `s2 == NULL`). Pendant le parcours, on peut retourner un résultat dès que les valeurs `Fils(n1)` et `Fils(n2)` sont différentes. `rc` contient alors la différence de la valeur des deux pointeurs.

   Si on arrive à la fin du `for`, trois cas de figure sont possibles:

   (a) `s1 == NULL` et `s2 != NULL`, ce qui signifie que $n1$ a moins de successeurs que $n2$ ($\#succ(n1) < \#succ(n2)$). Dans ce cas, on retourne arbitrairement la valeur '-1';

   (b) `s1 == NULL` et `s2 == NULL`, ce qui signifie que $n1$ et $n2$ ont exactement le même nombre de successeurs et qu'ils sont tous égaux ($succ(n1) = succ(n2)$). On a toujours `rc == 0` dans ce cas;

   (c) `s1 != NULL` et `s2 == NULL`, ce qui signifie que $n1$ a plus de de successeurs que $n2$ ($\#succ(n1) > \#succ(n2)$). Dans ce cas, on retourne arbitrairement la valeur '1'.

La fonction d'insertion d'un noeud dans une couche peut donc être modifiée. Pour la recherche d'un noeud équivalent, un algorithme standard de recherche dichotomique, utilisant `ComparerNoeuds` pour effectuer les comparaisons d'éléments du

```
int ComparerNoeuds(noeud *n1, noeud *n2)
{
  int rc;
  successeur *s1, *s2;

  rc = val(n1) - val(n2);
  if (rc != 0 )    /* Si les noeuds n´ont pas même valeur, */
    goto Exit;     /* on peut quitter */

  /* Parcours en parallèle des successeurs */
  for (s1 = PremierFils(n1), s2 = PremierFils(n2);
       s1 != NULL && s2 != NULL;
       s1 = FilsSuivant(s1), s2 = FilsSuivant(s2))
  {
    rc = (int)(Fils(s1) - Fils(s2));
    if ( rc != 0 ) /* Si on a trouvé des fils differents, */
      goto Exit;   /* on peut quitter */
  }

  /* On est à la fin des successeurs d´au moins un noeud */
  if (s1 == NULL)
  {
    if (s2 != NULL)
      rc = -1; /* n2 a plus de successeurs que n1 => n1 < n2 */
  }
  else
    rc = 1;    /* n1 a plus de successeurs que n2 => n1 > n2 */

Exit:
  return rc;
}
```

Algorithme 4.11: Fonction de comparaison de deux noeuds d'un Arbre Partagé

tableau, est appelé. Lorsque l'algorithme ne trouve pas de noeud équivalent, il retourne la position d'insertion du noeud dans le tableau de pointeurs. Il faut alors décaler les éléments à droite de cette position[8] et insérer la valeur du pointeur de noeud à l'emplacement devenu libre. Il faut également gérer l'allocation dynamique du tableau[9].

Le gain de performance dû à la nouvelle gestion des couches est d'autant plus important que le nombre de noeuds de valeur identiques est important (nombre de parcours parallèles des successeurs limité, grâce à la recherche dichotomique) et que le nombre de tentatives d'insertions est petit par rapport au nombre d'insertions effectives (coût réduit des décalages dans le tableau). Même dans le cas extrême où tous les noeuds d'une couche ont une valeur différente et où toutes les insertions sont effectives, le gain de performance est significatif, car la position d'insertion est trouvée très rapidement ($log_2(n)$) et le décalage des éléments du tableau n'est pas très coûteux. Un exemple de gains de performance dû à la nouvelle gestion des couches est présenté en Section 4.4.3.

4.4.2 Union incrémentale

Comme nous l'avons vu dans le Chapitre 4, beaucoup d'algorithmes dédiés aux Arbres Partagés ont besoin de l'opérateur ensembliste d'union. C'est le cas, par exemple, des algorithmes de calcul du produit accessible, de calcul d'accessibilité et de calcul d'un plus court chemin. Cependant, l'union effectuée n'est pas quelconque, car il s'agit toujours d'ajouter à un ensemble les éléments d'un autre ensemble. Par exemple, pendant le calcul du produit accessible, les nouveaux états calculés lors d'une itération sont ajoutés à l'ensemble des états déjà développés lors des itérations précédentes :

$$S \leftarrow S \cup NS$$

En pratique, ce genre d'instruction est implémenté de la manière suivante :

$$\left[\begin{array}{l} temp \leftarrow S \\ S \leftarrow temp \cup NS \\ Detruire(temp) \end{array}\right.$$

[8] D'un point de vue technique, cette opération n'est pas très coûteuse, car la plupart des bibliothèques 'C' contiennent une fonction de déplacement de zones mémoires. Cette fonction est en général écrite en assembleur et donc optimisée pour chaque type de processeur.

[9] De nouveau d'un point de vue technique, la taille allouée d'un tableau de pointeurs dans une couche est toujours la puissance de 2 supérieure ou égale au nombre de noeuds dans la couche. D'une part, cette méthode permet de ne pas réallouer trop souvent le tableau. D'autre part, le compilateur utilisé (`gcc`) spécifie que les demandes d'allocations mémoires les plus rapidement satisfaites sont celles qui requièrent un nombre d'octets égal à une puissance de 2.

On "gaspille" donc en moyenne $\frac{2^n - 2^{n-1}}{2} = 2^{n-2}$ éléments du tableaux (n étant le nombre de noeuds).

On doit donc, à chaque fois qu'une union est effectuée, créer un nouvel Arbre Partagé (S) représentant l'union des éléments de S et NS, puis détruire l'Arbre Partagé précédent (*temp*), même si NS ne contient que très peu d'éléments par rapport à S.

L'idée est de définir un nouvel algorithme d'union, appelé *union incrémentale*, qui modifie la structure interne d'un Arbre Partagé pour y ajouter les éléments d'un autre Arbre Partagé. Par exemple, l'algorithme va modifier l'Arbre Partagé représentant S pour y ajouter les éléments de NS. Bien entendu, le résultat de l'union incrémentale de S et NS doit donner exactement le même résultat que l'union classique. L'union incrémentale de S et NS est notée :

$$SelfUnion(S, NS)$$

L'algorithme d'union incrémentale fonctionne à peu près selon le même principe que l'algorithme d'union classique (Section 4.1.2). Les Arbres Partagés représentant S et NS sont parcourus en parallèle de haut en bas et les nouveaux noeuds sont ajoutés dans S de bas en haut.

Une première différence avec l'algorithme d'union classique est qu'il n'est pas nécessaire d'appeler récursivement l'algorithme pour des valeurs de noeud présentes dans S et pas dans NS. Ces noeuds représentent en effet des sous-arbres avec des éléments présents uniquement dans S, il est donc inutile de les modifier.

Un seconde différence est que la plupart des noeuds créés récursivement et ajoutés de bas en haut existent déjà dans l'Arbre Partagé représentant S. L'ajout d'un noeud dans une couche se résume donc souvent à rechercher s'il existe un noeud équivalent dans la couche[10] .

De plus, deux problèmes spécifiques se présentent :

- Lorsque, au retour d'un appel récursif sur l'union incrémentale, il faut ajouter un successeur à un noeud de S (ce successeur est la racine du sous-arbre résultat du dernier appel récursif), il est possible que le noeud de S possède déjà un successeur de même valeur, mais qui est la racine d'un sous-arbre différent. Ce cas de figure se présente, par exemple, lorsque de nouveaux éléments sont ajoutés dans un sous-arbre. Le nouveau sous-arbre, créé récursivement, contient alors à la fois les éléments existants de S et les nouveaux éléments (il n'est donc pas nécessaire d'effectuer une union des deux sous-arbres). La solution à ce problème est donc très simple : il suffit de changer la valeur pointeur vers le successeur existant (pointeur `Fils`) et d'y mettre la valeur du pointeur vers le nouveau successeur.

[10] Cette opération est devenue très rapide avec l'amélioration de la gestion des couches vue en section précédente.

- Le deuxième problème est lié au premier. Lorsque l'algorithme remplace un successeur par un autre, il se peut que l'ancien successeur devienne un noeud orphelin, c'est-à-dire un noeud qui n'est successeur d'aucun autre. Cette situation est aisément détectable, car chaque structure 'noeud' contient un champ spécial (`NombreParents`), qui indique le nombre de noeuds parents. Lorsque cette valeur vaut 0, le noeud est orphelin.

  Il n'est cependant pas possible de détruire les noeuds orphelins "en direct" lors du calcul de l'union incrémentale, car il se peut qu'un noeud ne soit orphelin que temporairement. Un noeud orphelin peut en effet devenir successeur d'un nouveau noeud lors d'une opération intermédiaire future.

  De plus, le noeud devenu orphelin peut avoir été stocké dans la table de hachage comme résultat d'une opération intermédiaire précédemment effectuée. Si on détruit ce noeud prématurément et qu'il redevient successeur d'une autre noeud, l'algorithme ne fonctionnera pas (il faudrait modifier dynamiquement la table de hachage en cours de calcul, ce qui serait très coûteux).

  La solution à ce problème est d'effectuer, une fois le calcul de l'union incrémentale complètement terminé, un parcours de tous les noeuds, couche par couche, de haut en bas, de l'arbre S et d'éliminer dans chaque couche les noeuds orphelins. Le parcours doit être effectué de haut en bas pour supprimer également les noeuds ayant comme seul parent un noeud orphelin. A chaque fois qu'un noeud orphelin est détruit, la valeur du champ `NombreParents` de tous ses successeurs est décrémentée. Il se peut alors que de nouveaux orphelins soit créés (et détruits lors du parcours de la couche suivante).

4.4.3 Exemple de gains de performance

La Figure 4.23 présente des statistiques à propos de chaque couche d'un des plus gros Arbre Partagé qui a été expérimenté. Il a été généré par l'algorithme de calcul de plus court chemin (`trace`) dans le cadre du système de transitions synchronisé présenté en Section 6.5. Il s'agissait en fait de rechercher un plus court chemin dans un arbre de transitions comportant environ $3 * 10^9$ transitions. Il n'existait pas de chemins, ce qui implique le nombre d'éléments de l'Arbre Partagé à la fin de la recherche est égal au nombre de transitions du système, car toutes les transitions ont été parcourues. On remarque la taille importante de cet arbre : 24 couches, environ 60 000 noeuds et 210 000 successeurs, le tout occupant environ 3 700 Ko de mémoire.

Le nombre d'octets perdus à cause des tableaux de pointeurs (Section 4.4.1) est 113324. Il s'agit en fait de la somme des octets perdus à chaque couche. Par

```
Real time: 107.965 secs, CPU time: 103.520 secs

Path tree (definitive):
=======================
3011510272 elements
24 layers, 60510 nodes, 210513 sons
3734440 bytes (113324 wasted bytes).
    Layer   1 :   126/128   node(s) (Insertions: 100.0% [  126/126  ])
    Layer   2 :   940/1024  node(s) (Insertions: 100.0% [  940/940  ])
    Layer   3 :  1374/2048  node(s) (Insertions:  81.2% [ 1374/1693 ])
    Layer   4 :  3693/4096  node(s) (Insertions:  80.4% [ 3693/4593 ])
    Layer   5 : 14536/16384 node(s) (Insertions:  61.2% [14536/23756])
    Layer   6 :  8668/16384 node(s) (Insertions:  32.4% [ 8668/26744])
    Layer   7 :  8672/16384 node(s) (Insertions:  30.2% [ 8672/28704])
    Layer   8 :  2782/4096  node(s) (Insertions:  25.4% [ 2782/10937])
    Layer   9 :  1964/2048  node(s) (Insertions:  13.6% [ 1964/14390])
    Layer  10 :  1282/2048  node(s) (Insertions:   8.6% [ 1282/14822])
    Layer  11 :  1234/2048  node(s) (Insertions:   7.5% [ 1234/16407])
    Layer  12 :  1161/2048  node(s) (Insertions:   7.2% [ 1161/16212])
    Layer  13 :  4676/8192  node(s) (Insertions:   7.1% [ 4676/65952])
    Layer  14 :  3680/4096  node(s) (Insertions:   5.6% [ 3680/65472])
    Layer  15 :  2808/4096  node(s) (Insertions:   5.1% [ 2808/55240])
    Layer  16 :   670/1024  node(s) (Insertions:   4.9% [  670/13752])
    Layer  17 :   348/512   node(s) (Insertions:   3.4% [  348/10318])
    Layer  18 :   241/256   node(s) (Insertions:   3.0% [  241/7904 ])
    Layer  19 :   237/256   node(s) (Insertions:   3.0% [  237/7771 ])
    Layer  20 :   824/1024  node(s) (Insertions:   2.5% [  824/33048])
    Layer  21 :   468/512   node(s) (Insertions:   2.0% [  468/23424])
    Layer  22 :    62/64    node(s) (Insertions:   1.4% [   62/4569 ])
    Layer  23 :    62/64    node(s) (Insertions:   1.4% [   62/4569 ])
    Layer  24 :     1/32    node(s) (Insertions:   0.8% [    1/126  ])
```

Figure 4.23: Exemple de résultat de l'union incrémentale

exemple, dans la couche 2 (`Layer 2`), on perd $(1024 - 940) * 4 = 336$ octets, '4' étant le nombre d'octets occupé par un pointeur.

Pour chaque couche, on retrouve le nombre de noeuds, immédiatement suivi de la taille réelle du tableau de pointeurs de noeuds. La valeur `Insertions` indique le pourcentage des tentatives d'insertion qui se sont soldées par une insertion effective. Les deux valeurs qui suivent (entre crochets) servent à établir ce pourcentage : il s'agit du nombre d'insertions effectives et du nombre de tentatives d'insertion.

Comme on peut le voir sur la ligne `Layer 1`[11], le nombre d'itérations de l'algorithme est 126 (puisqu'il y a 126 noeuds dans la couche 1). La valeur 128 immédiatement après 126 indique la taille réellement allouée du tableau

On peut faire les remarques suivantes :

1. La mémoire "gaspillée" par les tableaux de pointeurs n'est pas très importante (environ 3% de la taille totale de l'arbre).

2. L'efficacité de l'union incrémentale est parfaitement illustrée. A partir de la couche 9, le pourcentage d'insertions effectives devient négligeable par rapport au nombre de tentatives d'insertions. On économise donc un très grand nombre d'insertions, opération coûteuse dans un tableau de pointeurs. Par exemple, pour la couche 18, il y a eu seulement 241 insertions sur 7904 tentatives.

Expérimentalement, on constate que le temps de calcul de cet arbre est d'environ 1 minute 45 secondes. Si, dans un premier temps, on remplace l'union incrémentale par l'union classique, le temps de calcul passe à plus de 12 minutes. Ensuite, si on remplace les tableaux de pointeurs par des listes chaînées, le temps de calcul passe à plus de 6 heures (!).

Nous reviendrons dans le Chapitre 6 sur les différences de temps de calcul existants entres les différentes versions des algorithmes.

[11] La couche 0 n'est pas représentée, car elle contient uniquement le noeud $\top$.

Chapitre 5

Architecture de MEC 4

Dans ce chapitre, nous décrivons sommairement l'architecture de la dernière version du logiciel MEC: MEC 4. Cette version a été entièrement développée selon une architecture orientée objets, avec le langage C++. Nous expliquons surtout les parties spécifiques aux Arbres Partagés.

Le but des concepteurs de MEC 4 était de rendre possible l'intégration de nouveaux types de représentation des systèmes de transitions, en gardant la compatibilité avec l'ancienne représentation énumérative. La nouvelle structure de représentation à intégrer dans cette architecture est la représentation des systèmes de transitions synchronisés avec Arbres Partagés[1].

Malheureusement, l'architecture a été pensée avec la représentation explicite des automates comme base de réflexion, ce qui entraîne des difficultés d'intégration que nous allons tenter d'expliquer dans les sections suivantes.

Deux volets sont envisagés pour l'intégration des Arbres Partagés dans MEC 4:

- Dans un premier temps (Section 5.1), des nouvelles classes C++ pour la représentation des systèmes de transitions synchronisés avec Arbres Partagés sont implémentées. Pour rappel, un système de transitions contient un ensemble d'états et un ensemble de transitions, ainsi qu'un nombre quelconque de marques et de compteurs.

  Lorsque ces nouvelles classes (et toutes les primitives redéfinies) sont fonctionnelles, tous les algorithmes génériques du logiciel (voir Chapitre 3) sont applicables (via les primitives) sur les Arbres Partagés. Le gain se situe donc au niveau de l'encombrement mémoire, mais les temps de calcul dépassent ceux de MEC, à cause de la nature itérative de ces primitives.

[1] L'intégration d'une autre structure, les *BDD's* (*Binary Decision Diagrams* [3]), est également envisagée depuis un certain temps déjà.

- Dans un deuxième temps (Section 5.2), tous les algorithmes dédiés aux Arbres Partagés (voir Chapitre 4) sont intégrés dans l'architecture, sans interférer avec les anciens algorithmes, puisque ces derniers doivent toujours être applicables aux systèmes de transitions normaux.

  Un point délicat est l'intégration de l'algorithme de calcul des points fixes d'un système d'équations.

Une fois ces deux phases réalisées, MEC 4 avec Arbres Partagés est pleinement fonctionnel. Avec la même version du logiciel, l'utilisateur peut alors dynamiquement choisir, au moment du calcul du produit synchronisé, entre une représentation explicite ou une représentation avec Arbres Partagés des automates synchronisés.

Lors du calcul d'une propriété sur un automate, le logiciel exécute l'algorithme adapté au type de représentation de l'automate. Par exemple, le calcul de `src` sur un automate avec Arbres Partagés est effectué par l'algorithme décrit en Section 4.3.1.

5.1 Classes C++ des systèmes de transitions

On entend par "classes des systèmes de transitions" toutes les classes C++ du logiciel utilisées pour la représentation d'un système de transition et des objets associés (états, transitions, marques, compteurs)[2].

Les sous-sections suivantes passent en revue l'ensemble de ces classes, décrivent une partie des services offerts[3] et la manière dont ces services sont réalisables avec les Arbres Partagés.

5.1.1 Systèmes de transitions

La Figure 5.1 représente la hiérarchie des classes "systèmes de transitions" de MEC 4. Les classes abstraites[4] sont représentées dans un cadre pointillé, les autres classes, dans un cadre continu. La hiérarchie présentée contient les classes suivantes :

[2] L'architecture est évidemment plus complexe que celle présentée dans cette section, mais nous évitons volontairement d'entrer dans trop de détails.

[3] En pratique, tous ces services sont implémentés sous forme de "fonctions membres" virtuelles des différentes classes C++.

[4] Pour rappel, en C++, une classe abstraite est une classe dont la création d'instance est interdite.

Les classes prédéfinies "st", "st_simple" et "st_synchro"

Brièvement, ces trois classes abstraites définissent le comportement général d'un système de transitions, à savoir la création d'objets "états" (Section 5.1.2) et "transitions" (Section 5.1.3), et la gestion des d'objets "marques" (Section 5.1.4) et "compteurs" (Section 5.1.5). En pratique, seule l'interface des fonctions membres est définie, c'est-à-dire qu'aucune des fonctions n'a d'implémentation à ce niveau (fonctions abstraites).

La classe "st_synchro" est une spécialisation de la classe "st". L'interface est légèrement modifiée pour les primitives d'ajout de multi-états et multi-transitions. La classe "st_simple" est également une spécialisation de la classe "st". Les primitives d'ajout d'états et de transitions sont redéfinies.

Les classes "st_ithc", "st_ithc_simple" et "st_ithc_synchro"

Ces classes contiennent, par encapsulation, les tableaux de représentation explicite des automates (voir Chapitre 3).

La classe "st_ithc" contient les cinq tableaux de représentation des états et transitions (Section 3.1.1). Ces tableaux sont communs à la représentation des automates simples et synchronisés, ce que explique la présence de cette classe.

La classe "st_simple_ithc" hérite de "st_ithc" et ajoute les deux tableaux pour la représentation des noms d'états et de labels de transitions (Section 3.1.4).

La classe "st_synchro_ithc" hérite de "st_ithc" et ajoute les deux tableaux de hachage pour la représentation des multi-états et multi-transitions (Section 3.1.5).

La classe "st_synchro_ap"

La classe "st_synchro_ap" constitue une nouvelle classe du logiciel. Elle définit l'implémentation des systèmes de transitions synchronisés avec Arbres Partagés. Deux Arbres Partagés sont encapsulés dans la classe: un Arbre Partagé pour les multi-états et un Arbre Partagé pour les multi-transitions (Section 4.2). Les marques et compteurs avec Arbres Partagés (voir Section 5.1.4 et Section 5.1.5) sont également pris en charge dans "st_synchro_ap".

5.1.2 Etats

Un objet de type "etat" est utilisé pour le parcours *itératif* des états d'un système de transitions. Des primitives des manipulation de marques sont également définies, pour ajouter un état dans une marque, retirer un état d'une marque et tester la présence d'un état dans une marque.

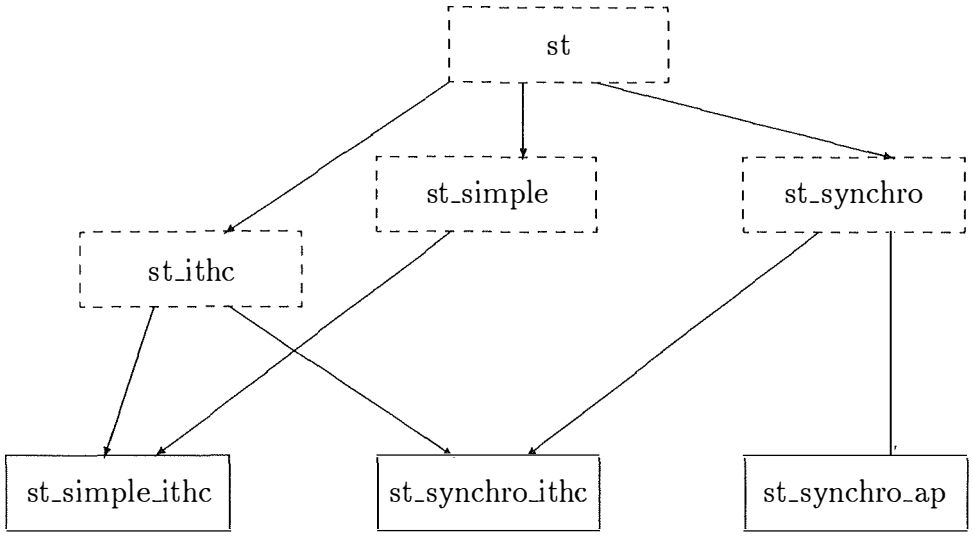


Figure 5.1: Hiérarchie des classes "systèmes de transitions"

La Figure 5.2 représente la hiérarchie simplifiée des classes "états". Cette hiérarchie est similaire à celle des systèmes des transitions. La sous-classe "`etat_synchro`" définit des primitives propres aux états d'un automate synchronisé. Par exemple, il existe une primitive pour retrouver les états composants du multi-état.

La classe "`etat_ithc`" et ses descendantes sont utilisées pour la représentation explicite. Elles contiennent, par encapsulation, l'indice de l'état dans les tableaux vus en Section 3.1.1.

La nouvelle classe "`etat_synchro_ap`" doit fournir tous les services de la classe "`etat_synchro`". Elle contient, par encapsulation, un n-uplet contenu dans l'Arbre Partagé des états d'un automate de type "`st_synchro_ap`".

Pour mieux comprendre l'utilisation de ces classes, la Figure 'Algorithme 5.1' donne un exemple de fonction C++ de parcours itératif des états de n'importe quel automate. Le paramètre "`automate`" est un pointeur sur un objet de type "`st`", qui peut être une instance de n'importe quelle classe dérivée de "`st`". Le second paramètre, "`argument`", est un pointeur objet de type "`marque`" (voir Section 5.1.4) indiquant quels états la fonction doit parcourir. Une précondition d'appel de cette fonction est que l'objet "`marque`" fasse partie des marques définies sur "`automate`".

La primitive "`un_etat`" demande à un objet automate la création d'un objet de type "`etat`". Par exemple, un automate de type "`st_synchro_ap`" crée un objet de type "`etat_synchro_ap`". La primitive "`nb_etats`" donne le nombre d'états contenu dans un automate.

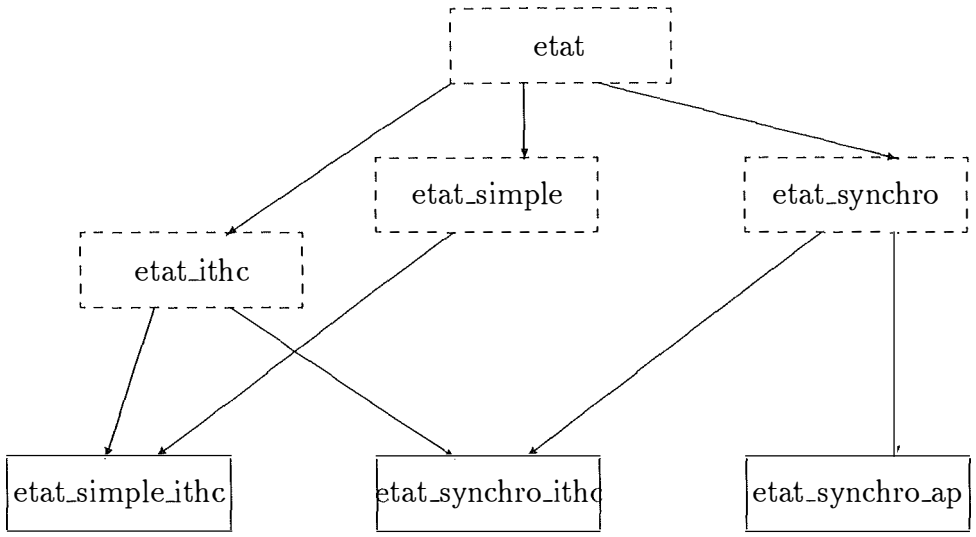


Figure 5.2: Hiérarchie des classes "état"

La fonction "`premier`" indique à l'objet "`et`" que le parcours itératif des états de l'automate commence. La fonction "`suivant`" demande à l'objet "`et`" de passer à l'état suivant de l'automate.

Si "`et`" est de type "`etat_ithc`", la primitive "`premier`" met à 0 l'indice contenu dans "`et`" ; la primitive "`suivant`" incrémente la valeur de cet indice.

Dans le cas d'un objet de type "`etat_synchro_ap`", le n-uplet contenu dans "`et`" est assigné au premier élément (lexicographiquement) de l'Arbre Partagé, c'est-à-dire à l'élément composé de la valeur des noeuds "les plus à gauche" de l'Arbre Partagé des états. Lors des appels de "`suivant`", le n-uplet prend la valeur des tous les éléments contenu dans l'arbre, par un parcours lexicographique de l'arbre (de gauche à droite, et de bas en haut).

Comme on le voit, l'implémentation des primitives avec les Arbres Partagés est plus complexe et requiert plus de temps de calcul. Nous ne détaillons pas autant les primitives dans les sections suivantes, mais le principe est toujours le même : la nature des primitives est très fortement liée à la représentation explicite, et leur implémentation avec des Arbres Partagés est plus complexe.

5.1.3 Transitions

Les classes "`transition`" sont définies selon le même principe que les classes "`etat`". La hiérarchie est cependant un peu plus complexe, car il existe deux types de transitions : les transitions directes, utilisées pour le parcours des transitions ayant un état donné comme source (chaînage direct), et les transitions inverses,

```
void ParcoursEtats(st *automate, marque *argument)
{
  etat *et;
  int i, nb;

  et = automate->un_etat();
  nb = automate->nb_etats();
  et->premier();
  for (i = 0; i < nb; i++ )
  {
    /* Traitement sur l´état ´et´ s´il est marqué ´argument´*/
    if ( et->tester_marque(argument) )
    {
      (...)
    }
    et->suivant();
  }
  delete et; /* détruire l´objet ´et´ créé par ´un_etat()´ */
}
```

Algorithme 5.1: Parcours itératif des états d'un système de transitions

utilisées pour le chaînage inverse. De plus, la notion de chemin est également définie via une classe "chemin_direct".

Une transition de type "trans_synchro_ap" contient un n-uplet de type 'transition', provenant de l'Arbre Partagé des transitions d'un automate de type "st_synchro_ap". Il est ainsi aisé d'obtenir l'état origine et l'état but de la transition, en effectuant une restriction sur la $1^{ère}$ ou sur la $3^{ème}$ valeur des éléments du n-uplet. Ces deux opérations constituent un exemple de primitives à réaliser.

Nous nous décrivons pas plus en détail les classes "transition". Signalons simplement que l'implémentation de ces classes a nécessité l'élaboration de toute une série d'algorithmes itératifs sur les Arbres Partagés.

5.1.4 Marques

La classe "marque" définit les opérations ensemblistes sur les marques d'états et de transitions. Les objets de la classe "marque_ithc", descendante de "marque", contiennent un tableau de bits pour représenter les sous-ensembles d'états ou de transitions (voir Section 3.1.2).

Chaque objet de la nouvelle classe "marque_ap" contient un Arbre Partagé représentant un sous-ensemble des états ou des transitions d'un automate de type "st_synchro_ap".

Les marques fournissent des services ensemblistes simples, tels que la copie, l'union, l'intersection, etc[5]. Par exemple, pour effectuer l'union de deux marques, on utilise une instruction du type:

```
resultat->is_union(arg1, arg2);
```

où resulat, arg1 et arg2 sont des objets de type "marque" (ou d'une classe descendante). Il s'agit d'assigner à resultat l'union des éléments contenu dans arg1 et arg2. Dans le cas de la représentation par tableau de bits, il suffit d'effectuer un OU binaire des tableaux de bits associés aux marques arguments. Dans le cas des Arbres Partagés, on appelle l'algorithme d'union des deux arbres contenus dans arg1 et arg2.

5.1.5 Compteurs

Nous ne nous attardons pas sur les compteurs, car ils ne sont plus utilisés pour le représentation avec Arbres Partagés. La Section 4.2.4 explique comment

[5] En réalité, le fonctionnement originel n'était pas aussi simple que celui présenté ici. Ce n'est qu'après avoir rencontré des problèmes pour l'intégration des Arbres Partagés que de nouvelles primitives ont été définies.

représenter un compteur avec des Arbres Partagés.

5.2 Classes C++ pour l'ajout d'algorithmes

Dans la section précédente, nous avons brièvement décrit l'architecture des classes utilisées pour la représentation des systèmes de transitions. Le point important à retenir est que tous les services offerts, à quelques exceptions près, par ces classes sont de nature 'itérative'. Une fois les classes, et toutes leurs primitives, implémentées, MEC 4 est capable d'appliquer tous les algorithmes vus au Chapitre 3 sur n'importe système de transitions, y compris un système représenté avec des Arbres Partagés.

Cependant, cette architecture n'est pas satisfaisante, car les temps de calcul sont beaucoup trop prohibitifs pour que le logiciel soit utilisable pratiquement sur des Arbres Partagés. Nous avons en effet vu au Chapitre 4 qu'il est beaucoup plus efficace, en général, de travailler de manière globale sur les Arbres Partagés. Cette section décrit comment tous les algorithmes présentés dans le Chapitre 4 sont intégrés dans l'architecture de MEC 4.

Le principal problème à résoudre est de permettre l'exécution d'algorithmes différents selon le type du système de transitions sur lequel un calcul est effectué. Les deux sous-sections suivantes décrivent les deux approches choisies pour résoudre le problème. La première approche concerne l'évaluation des opérateurs prédéfinis, et la seconde, l'évaluation des opérateurs définis comme solution d'un système d'équations.

5.2.1 Evaluation des opérateurs prédéfinis

Tous les opérateurs prédéfinis sont des objets de type "`fct_system`". Chaque objet contient des informations sur la nature de l'opérateur qu'il définit: nom, nombre et type des arguments, type du résultat. De plus, une fonction virtuelle d'évaluation contient le code nécessaire à l'évaluation de l'opérateur.

Dans la version originelle de logiciel, la fonction d'évaluation de chaque opérateur contient le code des algorithmes génériques (voir Chapitre 3). Par exemple, la fonction "`Evaluer`" de l'objet correspondant à la primitive `src` contient le code de l'algorithme vu en Section 3.2.2[6]. L'objet "`src`" définit également le nom de la primitive ("src"), le nombre d'argument (1), leur type ("transition") et le type du résultat ("état").

Tous les objets correspondant aux opérateurs prédéfinis sont identifiés par leur

[6]En réalité, la fonction contient une version C++ du code, qui est légèrement différente.

nom et sont stockés dans une liste globale. Lorsque le logiciel évalue une expression contenant un appel à un opérateur, il cherche dans la liste l'objet ayant le nom utilisé dans l'expression.

Une fois cet objet trouvé, des primitives d'accès permettent au logiciel de vérifier si le type et le nombre des arguments sont corrects. Si c'est le cas, les arguments sont évalués et stockés dans les marques "arguments" de l'objet. Ensuite, la fonction virtuelle d'évaluation est appelée. Son rôle est de stocker le résultat de l'évaluation dans la marque "résultat" de l'objet. Le logiciel peut alors accéder à cette marque et continuer l'évaluation de l'expression.

L'intégration des algorithmes dédiés aux Arbres Partagés consiste à modifier la fonction d'évaluation de tous les objets prédéfinis. Du code C++, ajouté au début de chaque fonction, est chargé de déterminer dynamiquement le type des marques "arguments". Si les marques sont de type "`marque_ap`", l'algorithme dédié aux Arbres Partagés peut être exécuté, avec les arbres des marques "arguments" comme paramètre. Au retour de l'algorithme, l'arbre résultant du calcul est transféré dans la marque "résultat".

La Figure 'Algorithme 5.2' contient un exemple de code utilisé pour le "court-circuitage" d'une fonction d'évaluation. La fonction `DOWNCAST(cls,obj)` permet de déterminer dynamiquement le type d'un objet. Si l'objet `obj` n'est pas de type `cls`, le résultat de l'expression est `NULL`. Si l'objet est de type `cls`, le résultat est un pointeur d'objet de type `cls`[7].

5.2.2 Evaluation des opérateurs définis comme solution d'un système d'équations

Un utilisateur du logiciel MEC peut définir de nouveaux opérateurs comme solution d'un système d'équations (voir Chapitre 2). Ces nouveaux opérateurs sont alors utilisables dans les expressions exactement de la même manière que les opérateurs prédéfinis. Leur évaluation est effectuée par un algorithme de calcul des points fixes d'un système d'équations.

D'un point de vue technique, la création d'un nouvel opérateur entraîne la création d'un objet de type "`fct_point_fixe`", qui dérive de "`fct_system`". En plus du nom de l'opérateur et des informations sur ses arguments (voir section précédente), chaque objet de ce type contient la définition du système d'équations (nombre,

[7]Pour les connaisseurs du langage C++, la fonction `DOWNCAST(cls,obj)` effectue une opération similaire à la fonction `dynamic_cast<cls,obj>` définie dans la norme "ANSI C++ 3.0". Malheureusement, le compilateur utilisé pour le développement de MEC 4 (`g++ 2.6.3`) ne respecte pas cette norme.

```
/* Teste si l´argument est une marque sur Arbre Partagé */
if ( DOWNCAST(marque_ap, argument) != NULL )
{
  /* Appel de l´algorithme dédié aux Arbres Partagés */
  (...)
}
else
{
  /* Algorithme `générique´ de MEC 4 */
  (...)
}
```

Algorithme 5.2: Exemple de court-circuitage de la fonction d'évaluation d'un opérateur

type et signe des variables, nombre et type des paramètres, et définition des termes).

Le problème de l'évaluation des nouveaux opérateurs est le même que pour les opérateurs prédéfinis : ils peuvent être évalués sur des systèmes de transitions normaux, ou sur des systèmes de transitions avec Arbres Partagés. Cependant, l'approche utilisée dans la section précédente n'est pas satisfaisante, car les modifications à effectuer dans la fonction d'évaluation seraient trop importantes. En effet, les algorithmes de calcul des point fixes sont plus complexes que les algorithmes de calcul des opérateurs prédéfinis.

La solution choisie, proposée par *Paul Crubillé*, consiste créer un objet de type "`algo_point_fixe`" par algorithme de résolution. Chaque objet contient une priorité qui définit l'ordre d'application des algorithmes sur un système d'équations. Lors d'une évaluation d'un opérateur, l'algorithme de priorité maximale est dans un premier temps appelé. Si celui-ci détecte qu'il ne peut pas s'appliquer aux système de transitions, il passe la main à l'algorithme de priorité immédiatement inférieure.

Dans la version actuelle du logiciel, deux objets "algorithmes" existent : le 1^{er}, de priorité minimale, contient l'algorithme MEC de résolution (voir Section 3.2.6) ; le second, de priorité supérieure, est l'algorithme dédié aux Arbres Partagés (voir Section 4.3.7). Lors d'une évaluation, ce dernier est appelé en priorité. S'il détecte un système de transitions avec Arbres Partagés, il résout le système d'équations avec des Arbres Partagés ; sinon, il passe la main à l'algorithme générique de MEC.

En conclusion à ce chapitre, nous avons montré quelques-unes des difficultés

rencontrées lors de l'intégration des Arbres Partagés dans MEC 4.

Chapitre 6

Résultats expérimentaux

Ce chapitre est dédié à la présentation de résultats expérimentaux. Après avoir défini le protocole de test, nous donnons les résultats mesurés sur quatre exemples d'automates synchronisés de natures très différentes. L'analyse de ces exemples montre que MEC/AP est quasiment toujours plus rapide et surtout moins gourmand en mémoire que MEC.

6.1 Protocole de test

Tous les tests présentés dans ce chapitre ont été exécutés sur un PC 486 DX4-100Mhz, avec 20 Mo de mémoire physique. Le système d'exploitation est Linux 1.3.2. Le compilateur utilisé est `g++` version 2.6.3.

Nous appelons "MEC" la version du logiciel utilisant la représentation explicite de automates et "MEC/AP" la version du logiciel utilisant les Arbres Partagés et les algorithmes dédiés. En pratique, une seule version du logiciel existe. Le choix d'une représentation explicite ou avec Arbres Partagés est fait par l'utilisateur au moment de la synchronisation des automates (voir Chapitre 5).

Toutes les tailles des objets en mémoire (Arbres Partagés, tableaux, etc.) sont exprimées en kilo-octets (Ko).

Tous les temps de calcul sont exprimés en secondes. En ce qui concerne MEC/AP, les temps indiqués ont été obtenus avec toutes les améliorations des algorithmes vues au Chapitre 4 :

- Pour le calcul du produit accessible, la deuxième version de l'algorithme est utilisée, c'est-à-dire que seuls les états accessibles sont calculés lors de la recherche du point fixe (voir Section 4.3.2).

- Les noeuds d'une couche sont triés et insérés via un tableau de pointeurs (voir Section 4.4.1).
- L'union incrémentale remplace l'union classique lorsque c'est possible (voir Section 4.4.2).

En plus du calcul du produit accessible, le calcul des propriétés suivantes a été systématiquement effectué[1] :

`dead := * - src(*)`
: Calcul de l'ensemble des états du système qui n'ont pas de successeurs.

`deadlock := inevitable(dead, *)`
: Calcul de l'ensemble des états du système qui mènent toujours à un état de `dead`. La fonction `inevitable` est une fonction définie comme plus petite solution du système d'équations suivant (définition MEC):

```
function inevitable(X : state; Y : trans)
  return Z : state;
var T : _trans
begin
Z = X \/ (* - src(T));
T = Y /\ rtgt(* - Z)
end.
```

Le calcul de `deadlock` permet de comparer l'efficacité de des deux algorithmes de recherche du point fixe d'un système d'équations (voir Section 3.2.6 et Section 4.3.7).

`trace := trace(initial, *, dead)`
: Calcul d'un plus court chemin entre `initial` et `dead`. Les temps de calcul mesurés avec MEC et MEC/AP peuvent être très différents à cause de la différence de stratégie des algorithmes: MEC part des états `dead` pour essayer de trouver un chemin menant à `initial`. MEC/AP fait exactement le contraire. Dans certains cas, l'ensemble `dead` ne contient aucun élément, rendant ainsi MEC/AP anormalement beaucoup plus lent que MEC.

`reach := reach(initial, *)`
: Calcul des états accessibles à partir de `initial`.

`coreach := coreach(dead, *)`
: Calcul des états co-accessibles à partir de `dead`.

[1] Ces propriétés correspondent à des calculs généraux, souvent effectués par les utilisateurs de MEC. Nous avons pris soin de tester tous les algorithmes dédiés aux Arbres Partagés (voir Chapitre 4).

`loop := loop(*,*)`
: Calcul des toutes les transitions faisant partie d'une composante fortement connexe du graphe de l'automate.

Une dernière remarque concernant les tests : lorsque le nombre d'éléments représenté par un Arbre Partagé est très élevé (à partir de plusieurs millions d'éléments), nous donnons une approximation limitée à trois chiffres significatifs de ce nombre. Par exemple, l'approximation du nombre "6 169 234" est notée "6.17e+06", qui représente le nombre "$6.17 * 10^6$".

6.2 *Schedulers* de Milner

L'exemple des *schedulers* de Milner est souvent utilisé dans la littérature pour comparer des algorithmes de calcul de produit accessible. Cet exemple est intéressant car le nombre d'états et de transitions du produit accessible croit exponentiellement avec le nombre d'automates.

Les automates composants un *scheduler* sont de deux types :

- Un automate *starter*, dont le rôle est d'activer le 1^{er} *cycler*.
- Plusieurs automates *cycler*. Un *cycler* attend d'être activé (transition `rc`) avant d'exécuter une action (transition `alp`). Ensuite, il peut activer le *cycler* suivant (transition `sc`) avant ou après avoir exécuté une action interne (transition `tau`).

La définition MEC des automates *starter* et *cycler* se trouve en Figure 6.1. La définition MEC de la contrainte de synchronisation pour un *scheduler* composé d'un *starter* et de deux *cyclers* se trouve en Figure 6.2. Les contraintes de synchronisation sont construites selon un principe identique lorsque le nombre de *cyclers* augmente.

Dans cette section, nous présentons d'abord une comparaison entre MEC et MEC/AP. Ensuite, nous présentons des résultats obtenus uniquement avec MEC/AP, car trop grands (gourmands en place mémoire) pour être traités par MEC.

6.2.1 Comparaison de MEC et MEC/AP

Le Table 6.1 présente une comparaison de MEC et MEC/AP du point de vue du temps de calcul et de la taille de l'automate synchronisé pour un *scheduler* composé de 6, 8, 10 et 12 *cyclers*.

```
transition_system  starter;
0  |- e  -> 0 ,
      go -> 1 ;
1  |- e  -> 1 ;
< initial = { 0 }  >.
```

```
transition_system  cycler;
0  |- e   -> 0 ,
      rc  -> 1 ;
1  |- e   -> 1 ,
      alp -> 2 ;
2  |- e   -> 2 ,
      sc  -> 3 ,
      tau -> 4 ;
3  |- e   -> 3 ,
      tau -> 0 ;
4  |- e   -> 4 ,
      sc  -> 0 ;
< initial = { 0 }  >.
```

Figure 6.1: Définition MEC des automates `starter` et `cycler`

```
synchronization_system  sched2
                    < width = 3 ;
                      list = (starter,cycler,cycler) > ;
(go      . rc    . e     ) ;
(e       . rc    . sc    ) ;
(e       . sc    . rc    ) ;
(e       . alp   . e     ) ;
(e       . e     . alp   ) ;
(e       . tau   . e     ) ;
(e       . e     . tau   ) .
```

Figure 6.2: Contrainte de synchronisation pour 2 *cyclers*

nombre de *cyclers*	6	8	10	12
$\#S^a$	577	3 073	15 361	73 729
$\#T^b$	2 017	13 825	84 481	479 233
temps MEC[c]	0.7	6.3	42.4	256.4
temps MEC/AP[d]	0.3	0.5	0.7	1.3
taille(S) MEC[e]	40.9	81.9	394.1	1 767.8
taille(S) MEC/AP[f]	3.4	4.5	5.6	6.7
taille(T) MEC[g]	61.9	522.9	2 729.1	11 989.2
taille(T) MEC/AP[h]	8.3	11.5	14.8	18.0

[a] Nombre d'éléments dans l'ensemble des états accessibles
[b] Nombre d'éléments dans l'ensemble des transitions accessibles
[c] Temps (en secondes) pour calculer le produit accessible, avec MEC
[d] Temps (en secondes) pour calculer le produit accessible, avec MEC/AP
[e] Taille approximative (en Ko) des tableaux représentant S
[f] Taille approximative (en Ko) de l'Arbre Partagé représentant S
[g] Taille approximative (en Ko) des tableaux représentant T
[h] Taille approximative (en Ko) de l'Arbre Partagé représentant T

Table 6.1: Comparaison de MEC et MEC/AP pour les *schedulers* de Milner

Au niveau de l'encombrement mémoire, nous voyons clairement que la taille des tableaux de MEC augmente exponentiellement avec le nombre de *cyclers*, tandis que la taille des Arbres Partagés de MEC/AP augmente linéairement. Sur l'ordinateur de test, il s'est avéré impossible, avec MEC, de calculer un *scheduler* avec plus de 12 *cyclers*, à cause de l'encombrement mémoire. Avec MEC/AP, nous sommes allés jusqu'à 60 *cyclers* (voir section suivante).

Au niveau des temps de calcul, MEC est linéaire sur le nombre d'états et de transitions de l'automate synchronisé (c'est-à-dire exponentiel sur le nombre de *cyclers*), tandis que MEC/AP semble être quadratique[2] sur le nombre de *cyclers*. Les temps de calcul sont très largement favorables à MEC/AP.

[2] D'une manière générale, il est assez difficile de faire une analyse précise des temps de calcul obtenus avec MEC/AP, vraisemblablement à cause de la gestion dynamique de la mémoire. Les algorithmes de MEC/AP sont en effet beaucoup plus dynamiques que les algorithmes de MEC, qui travaillent toujours dans les tableaux de représentation énumérative des états et transitions. De son côté, MEC/AP doit créer et détruire beaucoup de données temporaires, et utilise des tableaux de hachage pour accélérer les algorithmes (voir la Section 4.1.2 pour des précisions sur ces deux points).

nombre de *cyclers*	20	30	40	50	60
#vecteurs[a]	61	91	121	151	181
$\#R$[b]	2.61e+15	3.81e+22	4.96e+29	6.05e+36	7.09e+43
taille(R)[c]	36.0	54.9	73.7	92.5	111.4
temps(1) R[d]	0.8	1.8	3.7	5.6	8.4
temps(2) R[e]	0.4	0.9	1.7	2.7	4.2

[a]Nombre de vecteurs de synchronisation
[b]Nombre d'éléments dans la relation de transition globale
[c]Taille approximative (en Ko) de l'Arbre Partagé représentant R
[d]Temps (en secondes) pour calculer R avec l'union classique
[e]Temps (en secondes) pour calculer R avec l'union incrémentale

Table 6.2: Tests de performance de MEC/AP (calcul de R)

6.2.2 Test de performance de MEC/AP

Dans cette section, nous présentons un test de performance de MEC/AP sur l'exemple des *schedulers* de Milner. Le but est de voir comment se comporte l'algorithme de calcul du produit accessible en terme de temps de calcul et d'encombrement mémoire lorsque le nombre de *cyclers* augmente. Les tests effectués font varier le nombre de *cyclers* de 20 à 60, par pas de 10.

La Table 6.2 donne les temps de calcul de la relation de transition globale (R). Comme on le voit, ils sont négligeables par rapport aux temps de calcul du produit accessible (voir Table 6.3). L'algorithme d'union incrémentale étant utilisé pour le calcul de R, nous avons indiqué les temps avec union incrémentale et avec union classique. Lorsque l'union incrémentale est utilisée, les temps de calcul de R sont divisés par 2.

La Table 6.3 donne les temps de calcul du produit accessible. Ces temps semblent être quadratiques sur le nombre de *cyclers*.

6.2.3 Calcul des propriétés avec 10 *cyclers*

Les temps de calcul des propriétés vues en Section 6.1 sont repris dans la Table 6.4. L'automate synchronisé est un *scheduler* composé de 10 *cyclers*. Comme on peut le voir, MEC/AP est plus rapide que MEC pour quasiment tous les calculs effectués.

nombre de *cyclers*	20	30	40	50	60
#itérations[a]	117	177	237	297	357
$\#S$[b]	3.14e+07	4.83e+10	6.59e+13	8.44e+16	1.03e+20
$\#T$[c]	3.30e+08	7.49e+11	1.35e+15	2.15e+18	3.16e+21
temps(1) MEC/AP[d]	14.6	57.0	155.8	323.5	608.6
temps(2) MEC/AP[e]	7.2	26.5	70.5	141.4	268.1
taille(S) MEC/AP[f]	11.0	16.4	21.9	27.3	32.8
taille(T) MEC/AP[g]	30.9	47.1	63.3	79.4	95.6

[a]Nombre d'itérations de l'algorithme de calcul du produit accessible
[b]Nombre d'éléments dans l'ensemble des états accessibles
[c]Nombre d'éléments dans l'ensemble des transitions accessibles
[d]Temps de calcul (en secondes), avec la $1^{ère}$ version de l'algorithme
[e]Temps de calcul (en secondes), avec la $2^{ème}$ version de l'algorithme
[f]Taille approximative (en Ko) de l'Arbre Partagé représentant S
[g]Taille approximative (en Ko) de l'Arbre Partagé représentant T

Table 6.3: Tests de performance de MEC/AP (calcul du produit accessible)

Primitive	Résultat[a]	Temps MEC[b]	Temps MEC/AP[c]
sync	–	42.4	0.7
dead	0	0.9	0.0
deadlock	0	6.1	0.1
trace	0	0.1	1.1
reach	15 360	1.0	0.4
coreach	0	1.1	0.0
loop	84 480	5.4	3.0

[a]Nombre d'états ou de transitions résultant du calcul, sauf pour la synchronisation
[b]Temps de calcul (en secondes) de la primitive, avec MEC
[c]Temps de calcul (en secondes) de la primitive, avec MEC/AP

Table 6.4: Temps de calcul de propriétés avec 10 *cyclers*

Primitive	Résultat[a]	Temps MEC[b]	Temps MEC/AP[c]
sync	–	–	70.5
dead	0	–	0.1
deadlock	0	–	0.3
trace	0	–	125.3
reach	6.59e+13	–	65.2
coreach	0	–	0.0
loop	1.35e+15	–	373.0

[a]Nombre d'états ou de transitions résultant du calcul, sauf pour la synchronisation
[b]La taille de l'automate étant trop importante, aucun temps n'a été mesuré avec MEC
[c]Temps de calcul (en secondes) de la primitive, avec MEC/AP

Table 6.5: Temps de calcul de propriétés avec 40 *cyclers*

6.2.4 Calcul des propriétés avec 40 *cyclers*

Les temps de calcul des propriétés vues en Section 6.1 sont repris dans la table Table 6.5. L'automate synchronisé est un *scheduler* composé de 40 *cyclers*. La taille de l'automate synchronisé étant trop grande pour MEC, aucun temps n'a pu être mesuré pour cette version.

6.3 Compteur kilométrique

L'automate synchronisé "Compteur kilométrique" est composé de cinq automates identiques, modélisant chacun une variable dont les valeurs possibles varient de 0 à 9. La Figure 6.3 contient la définition MEC de cet automate, nommé `cpt10`. La valeur initiale de chaque variable est 0. Les transitions '`i`' incrémentent la valeur de la variable et la transition '`r`' réinitialise la valeur à 0.

L'idée est de modéliser un compteur dont la valeur initiale est 0 et la valeur finale, 99 999, en incrémentant la valeur des variables composantes de droite à gauche (00000, 00001, 00002, ..., 00009, 00010, 00011, ...). Le définition MEC de la contrainte de synchronisation se trouve en Figure 6.4.

Cet exemple est intéressant car l'automate synchronisé est un graphe 'linéaire' : il est constitué d'un long filament de l'état "00000" à l'état "99999". Les algorithmes dédiés aux Arbres Partagés vont dégénérer vers leur '*worst-case*' en terme de temps de calcul.

En effet, tous les algorithmes vus en Section 4.3 travaillent de manière globale sur les Arbres Partagés. Or, dans l'exemple d'automate synchronisé que nous considérons ici, "travailler de manière globale" se résume à ajouter ou à retirer un

```
transition_system cpt10;
0 |- e -> 0,
     i -> 1;
1 |- e -> 1,
     i -> 2;
2 |- e -> 2,
     i -> 3;
3 |- e -> 3,
     i -> 4;
4 |- e -> 4,
     i -> 5;
5 |- e -> 5,
     i -> 6;
6 |- e -> 6,
     i -> 7;
7 |- e -> 7,
     i -> 8;
8 |- e -> 8,
     i -> 9;
9 |- e -> 9,
     r -> 0;
< initial = { 0 } >.
```

Figure 6.3: Définition MEC de l'automate `cpt10`

```
synchronization_system s100000 < width = 5 ;
    list = ( cpt10, cpt10, cpt10, cpt10, cpt10 ) >;

( e . e . e . e . i );
( e . e . e . i . r );
( e . e . i . r . r );
( e . i . r . r . r );
( i . r . r . r . r ).
```

Figure 6.4: Contrainte de synchronisation pour le compteur kilométrique

Ensemble[a]	Nombre d'éléments	Taille MEC[b]	Taille MEC/AP[c]
R	99 999	–	5.4
S	100 000	2 509.8	3.3
T	99 999	2 910.9	5.4

[a] R est la relation de transition globale (calculé par MEC/AP uniquement), S, l'ensemble des états accessibles et T, l'ensemble des transitions accessibles

[b] Taille approximative (en Ko) des tableaux représentant l'ensemble S ou T

[c] Taille approximative (en Ko) de l'Arbre Partagé représentant l'ensemble R, S ou T

Table 6.6: Taille du produit accessible pour le compteur kilométrique

élément à la fois dans les Arbres Partagés intermédiaires. Par exemple, le nombre d'itérations l'algorithme de calcul du produit accessible est égal au nombre d'états de l'automate, un nouvel état étant créé à chaque itération.

La Table 6.6 donne une comparaison de l'encombrement mémoire du produit accessible avec MEC et MEC/AP. La Table 6.7 reprend les différents temps de calcul des propriétés vues en Section 6.1.

On peut remarquer qu'en terme d'encombrement mémoire, MEC/AP est beaucoup moins gourmand que MEC. Cela s'explique par le fait que les graphes des Arbres Partagés S et T sont quasi-complets.

Par contre, en terme de temps de calcul, MEC est plus stable et souvent beaucoup plus rapide que MEC/AP. Par exemple, le calcul de `deadlock` est extrêmement long avec MEC/AP, car chaque équation du système d'équations de la fonction `inevitable` doit être évaluée environ 100 000 fois avant que le point fixe soit atteint. De plus, le calcul de `trace` s'est avéré impossible à effectuer, à cause d'un temps de calcul trop élevé. Le plus court chemin contenant 99 999 transitions, l'algorithme de calcul de `trace` (voir Section 4.3) doit générer un Arbre Partagé intermédiaire ayant 99 999 noeuds dans la première couche, ainsi que maintenir tous les sous-arbres de ces 99 999 noeuds.

6.4 Algorithme d'exclusion mutuelle de Peterson

L'algorithme d'exclusion mutuelle de Peterson, vu au Chapitre 1, peut être généralisé à plus de 2 processus. L'algorithme modélisé dans cette section travaille avec 4 processus, 4 variables partagées "`Q`" et 3 variables partagées "`TURN`". Cet exemple est intéressant car les Arbres Partagés générés par la synchronisation sont peu économes en terme de mémoire occupée.

La Table 6.8 donne une comparaison de l'encombrement mémoire du produit accessible avec MEC et MEC/AP. La Table 6.9 reprend les différents temps de

Primitive	Résultat[a]	Temps MEC[b]	Temps MEC/AP[c]
sync	–	56.4	87.1
dead	1	0.6	0.0
deadlock	100 000	22.6	2 641.4
trace	99 999	4.2	–
reach	99 999	2.0	87.5
coreach	99 999	6.6	83.5
loop	0	19.1	791.2

[a]Nombre d'états ou de transitions résultant du calcul, sauf pour la synchronisation
[b]Temps de calcul (en secondes) de la primitive, avec MEC
[c]Temps de calcul (en secondes) de la primitive, avec MEC/AP

Table 6.7: Temps de calcul des propriétés du compteur kilométrique

Ensemble[a]	Nombre d'éléments	Taille MEC[b]	Taille MEC/AP[c]
R	603 979 776	–	257.7
S	36 592	854.5	236.8
T	111 974	3 285.5	1 338.4

[a] R est la relation de transition globale (calculé par MEC/AP uniquement), S, l'ensemble des états accessibles et T, l'ensemble des transitions accessibles
[b]Taille approximative (en Ko) des tableaux représentant l'ensemble S ou T
[c]Taille approximative (en Ko) de l'Arbre Partagé représentant l'ensemble R, S ou T

Table 6.8: Taille du produit accessible pour 4 processus de Peterson

calcul des propriétés vues en Section 6.1.

Nous pouvons constater dans un premier temps que le gain en occupation mémoire avec les Arbres Partagés est beaucoup moins important que dans les autres exemples traités dans ce chapitre. D'autant plus qu'il faut ajouter à la taille des Arbres Partagés, la taille des structures de données temporaires et des tables de hachage[3].

Les temps de calculs sont évidemment plus importants avec MEC/AP qu'avec MEC, à cause de la taille importante des Arbres Partagés.

6.5 Le protocole FRP/DT

Le protocole FRP/DT est un protocole permettant de faire varier dynamiquement les débits des circuits virtuels d'un réseau de type ATM (*Asynchronous Transfer*

[3]Pratiquement, cet exemple constitue le seul cas que nous avons expérimenté où l'occupation mémoire totale d'un processus "MEC/AP" est plus importante que celle d'un processus "MEC".

Primitive	Résultat[a]	Temps MEC[b]	Temps MEC/AP[c]
sync	–	116.1	56.5
dead	0	0.1	1.5
deadlock	0	9.2	29.7
trace	0	0.1	122.9
reach	36 592	1.2	34.2
coreach	0	2.4	0.2
loop	111 974	8.3	265.3

[a]Nombre d'états ou de transitions résultant du calcul, sauf pour la synchronisation
[b]Temps de calcul (en secondes) de la primitive, avec MEC
[c]Temps de calcul (en secondes) de la primitive, avec MEC/AP

Table 6.9: Temps de calcul des propriétés pour 4 processus de Peterson

Mode). Il a été modélisé et étudié dans [2]. Nous n'entrons pas dans les détails de la modélisation, car elle est assez complexe : l'automate synchronisé est composé de 22 automates. Parmi ces automates, certains modélisent le comportement des entités du réseaux, d'autres, les variables nécessaires au fonctionnement de ces entités, d'autres encore, des canaux à débit variable.

Le but de la modélisation était de mettre à jour et de déterminer la cause de certains problèmes apparus lors de l'implémentation du protocole dans les laboratoires de *France Telecom*. Signalons que le logiciel MEC a fait ses preuves, puisque certains problèmes ont pu être analysés en détail (voir [2])[4].

La première version de la modélisation, effectuée avec MEC, requiert le calcul d'un automate synchronisé ne contenant que quelques dizaines de milliers d'états et de transitions. Cette version limite fortement le nombre de canaux de communication et le nombre de valeurs de débits possibles sur ces canaux, à cause du problème de l'encombrement mémoire.

La version de la modélisation testée dans cette section, qui nous a grâcieusement été prêtée par *M.H. Skubiszewska*, constitue une version expérimentale, où le nombre de valeurs possibles des débits sur les canaux de communication est plus important. MEC/AP est la première version du logiciel MEC permettant de tester cet automate synchronisé. Décrire plus en détails la modélisation et les commandes nécessaires aux vérifications serait trop long, c'est pourquoi nous nous limitons au tests présentés en Section 6.1.

La Table 6.10 contient les informations concernant l'encombrement mémoire du produit accessible avec MEC/AP. La Table 6.11 reprend les différents temps de

[4]Parmi ces problèmes, citons brièvement une incohérence des débits sur les canaux, ou encore des demandes de changements de débits qui, dans certaines circonstances, ne sont jamais acceptées, ni refusées.

Ensemble[a]	Nombre d'éléments	Taille MEC[b]	Taille MEC/AP[c]
R	1.09e+12	–	291.3
S	7 432 704	–	28.8
T	3 011 510 272	–	414.4

[a] R est la relation de transition globale (calculé par MEC/AP uniquement), S, l'ensemble des états accessibles et T, l'ensemble des transitions accessibles
[b] La taille de l'automate étant trop importante, aucune taille n'a été mesurée avec MEC
[c] Taille approximative (en Ko) de l'Arbre Partagé représentant l'ensemble R, S ou T

Table 6.10: Taille du produit accessible pour l'automate FRP/DT

Primitive	Résultat[a]	Temps MEC[b]	Temps MEC/AP[c]
sync	–	–	43.2
dead	635 904	–	0.2
deadlock	1 790 464	–	18.5
trace	11	–	0.5
trace2	0	–	104.5
reach	7 399 936	–	40.6
coreach	6 796 800	–	5.4
loop	2 256 404 480	–	507.3

[a] Nombre d'états ou de transitions résultant du calcul, sauf pour la synchronisation
[b] La taille de l'automate étant trop importante, aucun temps n'a été mesuré avec MEC
[c] Temps de calcul (en secondes) de la primitive, avec MEC/AP

Table 6.11: Temps de calcul des propriétés pour l'automate FRP/DT

calcul des propriétés de la Section 6.1. Le calcul de la propriété `trace2` est effectué par la commande :

```
trace2 := trace(initial, * , {});
```

Cette propriété a été calculée pour déterminer le *worst-case*, en terme de temps de calcul, de l'algorithme de calcul de `trace`. Bien que le résultat soit un ensemble vide, puisqu'il n'existe aucun plus court chemin entre `initial` et `{}`, la taille l'Arbre Partagé intermédiaire généré par ce calcul est très importante (environ 3 700 Ko). On trouvera en Section 4.4.3 une analyse plus fine de cet Arbre Partagé.

Pour illustrer le comportement de l'algorithme de recherche des composantes fortement connexes d'un graphe (Section 4.3.6), la Table 6.12 reprend le temps de calcul et le nombre d'itérations lors de calculs successifs de `loop(*,*)`. Pour rappel, après avoir retirer les filaments du graphe, l'algorithme doit choisir une transition dans le graphe pour calculer les transitions accessibles et co-accessibles à partir

#Itérations	Temps(secondes)
177	507.3
113	431.8
181	503.2
101	391.7
117	410.8
201	512.6
89	384.2

Table 6.12: Divers temps de calcul de `loop(*,*)` sur l'automate FRP/DT

de cette transition. Dans certaines circonstances, ce choix est fait aléatoirement pour essayer de ne pas considérer les transitions sur un chemin de longueur 1.

Le nombre de grappes du graphe de l'automate synchronisé est 49. Le nombre d'itérations minimal théorique de l'algorithme est donc 49. On peut voir dans la table que le nombre d'itérations réel varie entre 89 et 201, tandis que le temps de calcul varie d'environ 6 à 9 minutes.

Chapitre 7

Conclusion

L'objectif de ce mémoire était de montrer comment les Arbres Partagés ont pu être intégrés dans le logiciel de vérification de modèle MEC. Les objectifs étaient en fait multiples, car au delà de la simple expérimentation, il fallait écrire une nouvelle version du logiciel qui soit plus efficace que la précédente, qui s'intègre dans l'ancienne version et qui fonctionne parfaitement.

La tâche n'était pas aisée, notamment à cause de problèmes techniques et du manque de souplesse de l'architecture orientée objets. De plus, il a été nécessaire de créer de nouveaux algorithmes dédiés aux Arbres Partagés pour les primitives prédéfinies et pour les primitives définies comme point-fixe d'un système d'équations. Le stage effectué à Bordeaux durant quatre mois a néanmoins été suffisant pour réaliser une version complète de la nouvelle version du logiciel: MEC/AP.

Pour un utilisateur habitué à MEC, MEC/AP s'utilise exactement de la même façon que son prédécesseur, mais le champ des problèmes traitables a grandi et les temps de calcul ont diminué.

Les résultats expérimentaux ont en effet montré que MEC/AP permet de gagner en mémoire occupée et en temps de calcul dans beaucoup de cas. Dans certains cas, les gains sont mêmes très importants (plusieurs ordres de magnitude). Ces résultats ont également eu le mérite de montrer que les algorithmes de gestion des Arbres Partagés avaient tendance à dégénérer avec la taille des Arbres Partagés. Ainsi, il a été possible d'améliorer ces algorithmes pour encore réduire les temps de calcul de manière significative.

Il s'est également avéré, dans un exemple précis de modélisation (l'algorithme de d'exclusion mutuelle de Peterson), que les Arbres Partagés ne permettent pas de représenter de manière très compacte les automates synchronisés. Par manque de recul et d'expériences supplémentaires, il est difficile de faire un lien entre la nature du problème traité et la taille des Arbres Partagés. De toute façon,

un Arbre Partagé étant toujours sous forme canonique, il n'est pas possible de résoudre ce problème quand il se présente.

Bien qu'en l'état actuel MEC/AP fonctionne, certains travaux futurs peuvent être envisagés, notamment pour accélérer encore les algorithmes. Il serait peut-être intéressant, par exemple, de créer une version "parallélisée" du logiciel (et des algorithmes de gestion des Arbres Partagés). Pour la création d'un Arbre Partagé, par exemple, plutôt que de parcourir séquentiellement tous les successeurs et ainsi créer séquentiellemnt les sous-arbres, on pourrait créer une unité d'exécution (un "*thread*") par successeur à parcourir. Ou encore, pour l'évaluation d'une expression MEC, il serait possible d'évaluer en parallèle les sous-arbres de l'expression (l'expression à gauche et à droite d'un opérateur ensembliste binaire, par exemple).

L'algorithme de calcul des points fixes peut lui-aussi certainement être amélioré, car une équation est systématiquement réévaluées lorsque la valeur de sa variable doit être calculée. Une solution (proposée par B. Le Charlier) serait peut-être de créer une table de hachage par équation, conservant les résultats des opérations antérieures et permettant ainsi la création de nouveaux algorithmes incrémentaux.

Enfin, un autre amélioration intéressante serait la réécriture de l'architecture de MEC 4 pour rendre l'intégration d'algorithmes ensemblistes plus aisée. Par exemple, le logiciel permet de définir des systèmes de transitions synchronisés dont les composants sont eux-mêmes des systèmes synchronisés. L'idée est de diminuer la complexité de modélisation de gros systèmes. Malheureusement, en pratique, cette possibilité n'est pas utilisable avec des Arbres Partagés, car la communication entre un système de transitions synchronisé et ses composants est assurée uniquement par des primitives itératives, rendant ainsi les temps de calcul prohibitifs.

D'un point de vue personnel, le stage effectué à Bordeaux et la rédaction de ce mémoire m'ont permis de m'adonner à mon activité favorite: le programmation. J'espère que le lecteur aura partagé mon enthousiasme et je suppose qu'il aura compris à quel point il a été intéressant pour moi de développer des algorithmes d'une telle complexité.

Bibliographie

[1] A. Arnold, D. Bégay & P. Crubillé. *Contruction and analysis of transition systems with MEC.* AMAST Series in Computing, 3, Wold Scientific, 1994.

[2] D. Bégay & M.-H. Skubiszewska. *Modélisation et vérification d'un protocole FRP/DT dédié au service audiotel*, Université de Bordeaux I (France), 1995.

[3] R.E. Bryant. *Graph-based algorithms for boolean function manipulation.* IEEE Transactions on Computers, 35(8), 1986.

[4] P. Crubillé. *Réalisation de l'outil MEC: Spécification fonctionnelle et architecture.* Université de Bordeaux I (France), 1989.

[5] B. Le Charlier. *Communications personnelles.* Décembre 1995.

[6] B. Le Charlier & P. Van Hentenryck. *A universal top-down fixpoint algorithm.* RP 93/22, Institute of Computer Science, Namur (Belgium).

[7] R. Milner. *Communication and concurrency.* Prentice Hall International.

[8] R. Paquay & D. Zampuniéris. *MEC with Sharing Trees.* Paper submitted to the International Test Conference, 1996.

[9] G. Perterson. *Myths about the mutual exclusion problem.* Information Processing Letters, 12(3), 1981.

[10] D. Zampuniéris & B. Le Charlier. *An efficient algorithm to compute the synchronized product.* International Workshop MASCOTs'95.

[11] D. Zampuniéris & B. Le Charlier. *Efficient handling of large sets of tuples with sharing trees.* International IEEE Conference DCC'95.